Resource Tracking in Parallel and Distributed Applications

Niels Drost, Rob V. van Nieuwpoort, Jason Maassen, and Henri E. Bal
Department of Computer Science, VU University
Amsterdam, The Netherlands
niels, rob, jason, bal @cs.vu.nl
www.cs.vu.nl/ibis

ABSTRACT

In this paper, we introduce the Join-Elect-Leave (JEL) model, a simple yet powerful model for tracking the resources participating in an application. This model is based on the concept of signaling, i.e., notifying the application when resources have *Joined* or *Left* the computation. In addition, the model includes *Elections*, which can be used to select resources with a special role. JEL supports several consistency models and is suitable for resource coordination of a wide variety of applications, ranging from the traditional fixed resource sets used in MPI, to flexible grid-oriented programming models.

Categories and Subject Descriptors:
C.2.1 [Computer-Communication Networks]: [Distributed Systems]: Distributed Applications

General Terms: Algorithms, Design

Keywords: Resource Tracking, Robust, Programming Models

1. INTRODUCTION

Traditionally, supercomputers and clusters are the main computing platforms used to run high performance parallel computations. In such systems, the set of resources used for an application usually is static. When a *job* is scheduled and started, it is assigned a number of machines (*nodes*), and it uses these resources until it finishes the computation.

In recent years, grid computing [2], peer-to-peer systems [1] and desktop grids [5] have become viable alternatives for running parallel applications. When moving to such systems, resource allocation is no longer static and resources may not always be available for the entire lifetime of the job. Therefore, *malleability* [7], the capability to handle changes in the resources used during a computation, and *fault tolerance* are needed to successfully run parallel and distributed applications in these environments.

Traditionally, parallel programming models such as MPI [3] use *ranks* to keep track of resources. Each node in the job is assigned a rank, and these are used as identifiers. However, the rank model is not able to handle changes in the resources, which may lead to gaps in the used ranks, or ranks being reused. As a result, ranks can no longer be used to reliably identify nodes, which may not be tolerated by the application. Therefore, programming models which use ranks are not suitable for the highly dynamic grid and peer-to-peer environments. Although work has been done to address these problems [4], fault tolerance remains largely unaddressed.

HPDC'08, June 23–27, 2008, Boston, Massachusetts, USA.
ACM 978-1-59593-997-5/08/06.

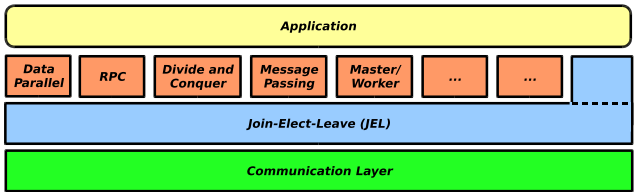

Figure 1: Layered overview of distributed system software including JEL

2. THE JOIN-ELECT-LEAVE MODEL

As an alternative model for tracking resources of parallel and distributed applications, we introduce the Join-Elect-Leave (JEL) model. JEL is a simple yet powerful model, which is able to reliably track resources in grid systems. As JEL is very flexible, it can adapt to the needs of different applications. Since it is not always practical or desirable to rewrite an application to use a new programming model, JEL can also be used to implement programming models. This eases the development of multiple programming models considerably. We refer to the programming models and applications together as *users*. Figure 1 shows an overview of JEL applications. JEL is implemented on top of a communication layer, for instance Sockets, or our Ibis [6] communication library. The programming models built on JEL thus use JEL to track the resources used, and the communication layer for sending data between nodes.

JEL has been specifically designed to cover the required functionality of all programming models typically found in distributed systems. Figure 1 lists the programming models we tested JEL on. In addition to these, JEL is generic enough to support other programming models, such as workflow systems and streaming frameworks.

Figure 2 shows a simplified version of the JEL API. The actual API used in our implementation also supports timeouts and throws exceptions where applicable. Parts of the API can be disabled by the user, and several consistency models are available, leading to a very flexible system.

In the JEL model, Users get notified whenever a new node joins a computation, or *pool*. Each node in the pool is assigned a unique identifier, which is included in the join notification. The identifier also acts as the contact address of the node, so the user can send messages to the new node. When a node joins a computation, it also gets notified of any nodes already present in the pool via the same notifications.

When a node notifies JEL that it is leaving the computation, the remaining nodes in the pool are notified of this fact using a *leave* notification. If a node does not leave gracefully, but crashes or

```
interface JEL {
    void init(Consistency electionConsistency,
              Consistency joinLeaveConsistency);

    Identifier join(String poolName);

    Identifier elect(String electionName);
    Identifier getElectionResult(String electionName);

    void declareDead(Identifier identifier);
    void leave();
}

//interface for notifications, called by JEL
interface JELNotifications {
    void joined(Identifier identifier);
    void left(Identifier identifier);
    void died(Identifier identifier);
}
```

Figure 2: JEL API (pseudocode, simplified)

is killed, the notification will consist of a *died* message instead. Implementations of JEL try to detect failing nodes, but the user can also report failures to JEL using the *declareDead* call.

JEL offers several consistency models for notifications. The *reliable* consistency model ensures that all notifications arrive at all nodes. In addition, they are guaranteed to arrive in the same order on all nodes. The *unreliable* consistency model has no such guarantees. Join and Leave notifications may get lost, or arrive out of order. As a third options, join and leave notifications can be turned off completely. Relaxing the consistency model of JEL allows greater flexibility when implementing it. For instance, we have implemented a fully distributed version of JEL that does not have any centralized components

To be able to select a single resource from a number of resources in a pool, JEL supports *Elections*. Each election has a unique name, and nodes can nominate themselves by calling a function with the election name as a parameter. The winner of the election will then be returned. Nodes can also retrieve the result without being a candidate.

Elections are not necessarily democratic. It is up to the implementation of the model to select a winner from all the candidates. An implementation may, for instance, simply select the first candidate as the winner. Like the notification part of JEL, elections also support multiple consistency models. In the master-worker model, *uniform* elections are used, which guarantees each election has a *single* winner, known at all nodes.

In addition to the *uniform* elections, JEL therefore supports *non-uniform* elections which have a relaxed consistency model. In the *non-uniform* model, nodes may have a different perception on the winner of an election. Instead an election is only guaranteed to converge to a single winner in unbound time. The implementation of JEL will try to get a consensus on the winner of an election as soon as possible, but in a large system it may take some time to reach an agreement among all the nodes. Until that time, nodes may perceive different winners for a single election.

As with the notifications, a more relaxed consistency model for elections provides more flexibility for their implementation. Although non-uniform elections may not be sufficient for all applications, they are suitable in many cases. In general, systems using central mechanisms to perform optimizations can usually be adapted to handle multiple managers, at the cost of efficiency.

	No Joins/Leaves	Unreliable Joins/Leaves	Reliable Joins/Leaves
No Elections	RMI		MPI, PVM Phoenix [4]
Non-Uniform Elections		Satin [7] (assigned master)	
Uniform Elections	Master-Worker workflow	Satin [7] (elected master)	DHT streaming data

Table 1: Overview of applicability of JEL in various programming models

3. PROGRAMMING MODELS

As said, JEL has specifically been designed to support all common programming models in use today. Table 1 gives an overview of the requirements of a number of programming models. For instance, RMI, Java's RPC-like model, is depicted to use neither elections nor joins and leaves. Since RMI requires users to identify machines explicitly, it does not require any features of JEL. Also listed in the table is Satin [7], a divide-and-conquer model.

4. CONCLUSIONS

In this paper we presented JEL: a flexible and robust model for tracking resources. Although the model is extremely simple, it supports both traditional programming models such as MPI, and more flexible grid oriented models like Satin. Multiple implementation strategies are possible, including fully distributed implementations.

5. ACKNOWLEDGMENTS

This work was carried out in the context of the Virtual Laboratory for e-Science project (www.vl-e.nl). This project is supported by a BSIK grant from the Dutch Ministry of Education, Culture and Science (OC&W) and is part of the ICT innovation program of the Ministry of Economic Affairs (EZ). This work has been supported by the Netherlands Organization for Scientific Research (NWO) grant 612.060.214 (Ibis: a Java-based grid programming environment).

6. REFERENCES

[1] N. Drost, R. V. van Nieuwpoort, and H. E. Bal. Simple locality-aware co-allocation in peer-to-peer supercomputing. In *Proc. of GP2P: Sixth International Workshop on Global and Peer-2-Peer Computing*, Singapore, may 2006.

[2] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the Grid: Enabling scalable virtual organizations. *Lecture Notes in Computer Science*, 2150, 2001.

[3] MPI forum website. http://www.mpi-forum.org/.

[4] K. Taura, K. Kaneda, T. Endo, and A. Yonezawa. Phoenix : a parallel programming model for accommodating dynamically joining resources. In *PPoPP'03: Proc. of the 9th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, 2003.

[5] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: the Condor experience. *Concurrency - Practice and Experience*, 17(2-4), 2005.

[6] R. V. van Nieuwpoort, J. Maassen, R. Hofman, T. Kielmann, and H. E. Bal. Ibis: an efficient Java-based grid programming environment. In *Joint ACM Java Grande - ISCOPE 2002 Conference*, Seattle, Washington, USA, November 2002.

[7] G. Wrzesinska, J. Maassen, and H. E. Bal. Self-adaptive Applications on the Grid. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'07)*, San Jose, CA, USA, March 2007.