IBIS: Real-Word Distributed Computing With IBIS

Bishal Kr. Jaiswal
School of Computer & System Science
Jaipur National University
Jaipur, India
Bishal14339@yahoo.com

***Abstract*—Ibis is an open source software framework that drastically simplifies the process of programming and deploying large-scale parallel and distributed grid applications. Ibis supports a range of programming models that yield efficient implementations, even on distributed sets of heterogeneous resources. Also, Ibis is specifically designed to run in hostile grid environments that are inherently dynamic and faulty, and that suffer from connectivity problems.**

I. INTRODUCTION

During recent years there is a big advancement in high performance distributed computer system in science and industry. These systems can provide transparent and efficient computing. For more complex applications existing scheduling system is limited. IBIS allows easy programming and development of distributed applications, even for dynamic, faulty and heterogeneous environment. IBIS is used in Multimedia computing, Spectroscopic data processing, Human brain scan analysis and Automatic grammar learning. The past two decades have seen tremendous progress in the application of high-performance and distributed computer systems in science and industry. Among the most widely used systems are commodity compute clusters, large-scale grid systems, and, more recently, economically driven computational clouds and mobile systems. In the last few years, researchers have intensively studied such systems with the goal of providing transparent and efficient computing, even on a worldwide scale.

In itself, each cluster, grid, and cloud provides well-defined access policies, full connectivity, and middleware that allow easy access to all its resources. Such systems are often largely homogeneous, offering the same software configuration or even the same hardware on every node. Combining several systems, however, is apt to result in a distributed system that is heterogeneous in software, hard-ware, and performance. This may lead to interoperability problems. Communication problems are also probable due to firewalls or network address translation (NAT), or simply because the geographic distance between the resources makes efficient communication difficult. Moreover, a combination of systems is often dynamic and faulty, as compute resources can be added or removed or even crash at runtime. The use of inherently heterogeneous and unreliable resources such as desktop grids, stand-alone machines, and mobile devices exacerbates these issues. We ascribe this rather limited use of grid systems to the intrinsic complexity of writing and deploying distributed applications. Grid programmers often are required to use low-level programming interfaces that change frequently. Programmers also must deal with system- and software-heterogeneity, connectivity problems, and resource failures. Also, managing a running application is complicated, because the execution environment may change dynamically as resources come and go all of these problems limit the acceptance of grid computing technology.

The ibis project aims to overcome these problems and to drastically simplify the programming and deployment process of (high-performance) grid applications. The Ibis philosophy is that grid applications should be developed on a local workstation and simply be launched from there. This write-and-go philosophy is directly derived from the abovementioned promise of the grid it requires minimalistic assumptions about the execution environment, and expects most of the environment's software to be sent along with the application. To this end, Ibis exploits Java virtual machine technology, and uses middleware-independent Application Programming Interfaces that are automatically mapped on to the available middleware.

II. REAL-WORLD DISTRIBUTED COMPUTING

An ad-hoc collection of compute resources that communicate with one another via some network connection constitutes a real-world distributed system. Writing application of such system is difficult. High performance distributed system can be abstracted complexities by a single software system that applies to any real world distributed system. IBIS allows easy programming and Development of distributed application, even for dynamic faulty and heterogeneous environment.

The uptake of high-performance distributed computing can be enhanced if these complexities are abstracted away by a single software system that applies to any real-world distributed system. Conceptually, such a

system should offer two logically independent subsystems: a programming system, offering functionality traditionally associated with programming languages and communication libraries; and a deployment system, offering functionality associated with operating systems. The programming system should allow applications to be not only efficient but also robust by providing programming models that offer support for fault tolerance and malleability adding and removing machines on the fly and that automatically circumvent any connectivity problems.

The deployment system should allow for easy deployment and management of applications, irrespective of to serve the vast majority of users, ranging from system and application-level software developers to application users, the subsystems should follow a layered approach, with programming interfaces defined at different abstraction levels, each tailored to different user needs.

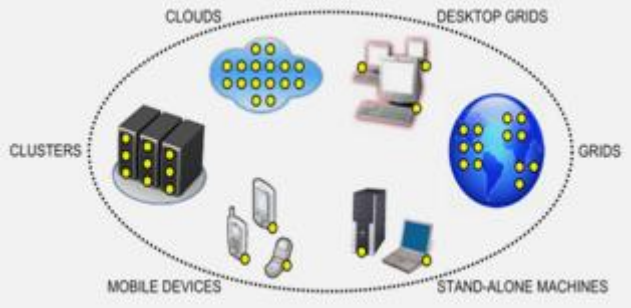

Fig 1:- A real-world distributed system consisting of clusters, grids, and clouds as well as desktop grids, stand-alone machines, and mobile devices. Clusters, grids, and clouds are well-organized subsystems that use their own middleware, programming interfaces, access policies, and protection mechanisms.

III. THE IBIS SYSTEM

The Ibis project aims to provide transparent solutions to all inherent grid computing problems, and as to make significant steps forward towards realizing the 'promise of the grid'. To this end, Ibis provides a rich software stack that provides all functionality that is traditionally (e.g., for sequential computers) associated with programming languages on the one hand, and operating systems on the other. More specifically, Ibis offers an integrated, layered solution, consisting of two subsystems: the High-Performance Application Programming System and the Distributed Application Deployment System (see Figure 2).

A. The Ibis High Performance Application Programming System

The Ibis High-Performance Application Programming System consists of (1) The IPL, (2) The programming models, and (3) Smart Sockets, described below.

1) The Ibis Portability Layer (IPL):- The IPL is at the heart of the High-Performance Application Programming System. It is a communication library that is written entirely in Java, so it runs on any platform that provides a suitable Java Virtual Machine (JVM). The library is typically shipped with the application (as Java jar files), such that no preinstalled libraries need to be present at any destination machine.

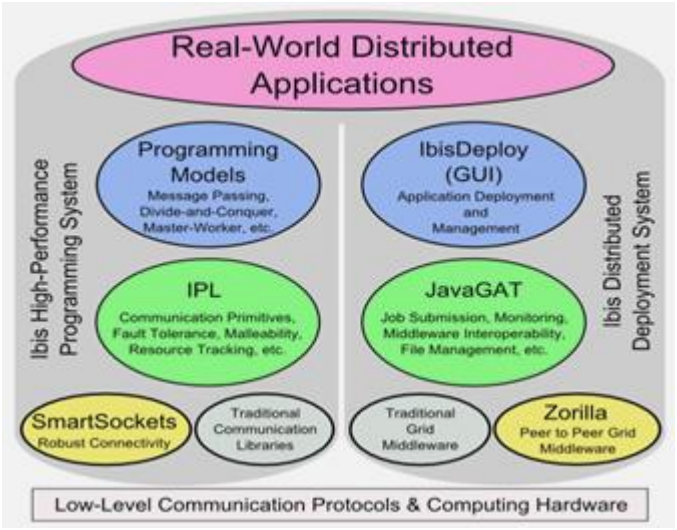

Fig 2 :- High Level Overview

The IPL provides a range of communication primitives (partially comparable to those provided by libraries such as MPI), including point-to-point and multicast communication, and streaming. It applies efficient protocols that avoid copying and other overheads as much as possible, and uses byte code rewriting optimizations for efficient transmission. To deal with real-world grid systems, in which resources can crash, and can be added or deleted, the IPL incorporates a runtime mechanism that keeps track of the available resources. The mechanism, called Join- Elect-Leave (JEL), is based on the concept of signaling, i.e., notifying the application when resources have Joined or Left the computation. JEL also includes Elections, to select resources with a special role.
The IPL has been implemented on top of the socket interface provided by the JVM, and on top of our own Smart Sockets library (see below). Irrespective of the implementation, the IPL can be used 'out of the box' on any system that provides a suitable JVM.

2) Ibis Programming Model :- The IPL has been used directly to write application but Ibis also provides several higher-level programming models on top of the IPL, including (1) an implementation of the MPJ standard. An MPI version in Java, (2) Satin a divide and conquer model described below the RMI, (3) Remote Method Invocation (RMI) is an object-oriented from of Remote Procedure call of GMI. (4)Group Method Invocation (GMI) is a generalization of RMI to group

communication, (5) Maestro a fault-tolerant and self optimizing data-flow model and (6) Jorus a user transparent parallel programming model for multimedia applications.
The most transparent model of these is Satin a divide and conquer system that automatically provides fault tolerance and malleability. Satin recursively splits a program into subtasks, and then waits until the subtasks have been completed. At runtime a Satin application can adapt the number of nodes to the degree of parallelism, migrate a computation away from overloaded resources, remove resources with slow communication links, and add new resources to replace resources that have crashed. As such, Satin is one of the few systems that provides transparent programming capabilities in dynamic systems.

3) Smart Socket :- To run a parallel application on multiple grid resources, it is necessary to establish network connections. In practice, however, a variety of connectivity problems exists that make communication difficult or even impossible, such as firewalls, Network Address Translation (NAT). It is generally up to the application user to solve such connectivity problems manually .The Smart Sockets library aims to solve connectivity problems automatically, with little or no help from the user. Smart Sockets integrates existing and novel solutions, including reverse connection setup, STUN, TCP splicing, and SSH tunneling. Smart Sockets creates an overlay network by using a set of interconnected support processes, called hubs. Typically, hubs are run on the front-end of a cluster. Using gossiping techniques, the hubs automatically discover to which other hubs they can establish a connection.

B. The Ibis Distributed Application Deployment System

The Ibis Distributed Application Deployment System consists of a software stack for deploying and monitoring applications, once they have been written. The software stack consists of (1) the Java GAT, (2) Ibis Deploy, and (3) Zorilla, described below.

1) The Java Grid Application Toolkit (JavaGAT):- Today, grid programmers generally have to implement their applications against a grid middleware API that changes frequently, is low-level, unstable, and incomplete. The JavaGAT solves these problems in an integrated manner. JavaGAT offers high-level primitives for developing and running applications, independent of the middleware that implements this functionality. It does so by integrating multiple middleware into a single coherent system. Extensions are frequently added by ourselves and others. JavaGAT further defines a powerful API for middleware developers to allow experiments with various middleware designs without hindering the application programmers. JavaGAT allows grid applications to transparently access remote data and spawn off jobs. It also provides support for monitoring, steering, user authentication, resource discovery, and storing of application-specific data. JavaGAT calls are dynamically forwarded to one or more middleware that implement the requested functionality.

The JavaGAT doesn't provide a new user/key management infrastructure. Rather, its security interface provides generic functionality to store and manage security information such as usernames and passwords. Also, the JavaGAT provides a mechanism to restrict the availability of security information to certain middleware systems or remote machines. For example, JavaGAT supports file operations through Grid FTP, SSH, HTTP, and SMB/CIFS, and resource management with GRMS, Globus (/WS) GRAM, SSH, PBS, SGE, Pro Active, and Zorilla. If a grid operation fails, it automatically dispatches the API call to an alternative middleware.

2) Ibis Deploy :- Many applications use the same deployment process. Therefore, Ibis Deploy provides a simple and generic API and GUI that can automatically perform commonly used deployment scenarios. For example, when a distributed Ibis application is running, Ibis Deploy starts the Smart Sockets hub network automatically. It also automatically uploads the program codes, libraries, and input files (*pre staging*) and automatically downloads the output files (*post staging*).

The Ibis Deploy GUI, shown in Figure 3, lets a user start, monitor, and stop applications in an intuitive manner and run multiple distributed applications concurrently. The user can add resources to a running application by simply providing contact information such as a host address and user credentials. This information can be reused in later experiments.

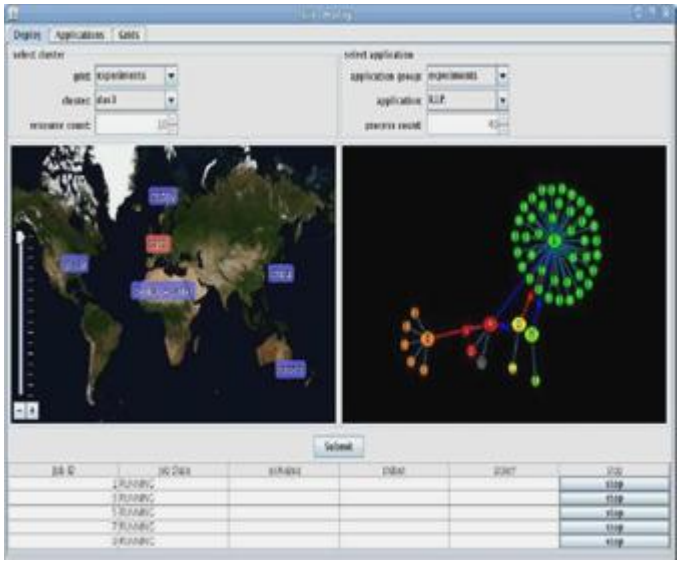

Fig 3:- The Ibis Deploy GUI that enables runtime loading of grids and applications (top), and keeping track of running processes (bottom).Center left shows the locations of available resources center right shows Smart Sockets connections between these resources.

3) Zorilla :- Most existing middleware APIs lack co-scheduling capabilities and don't support fault tolerance and malleability. To overcome these problems, Ibis provides Zorilla, a lightweight P2P middleware that runs on any real world distributed system. In contrast to traditional middleware, Zorilla has no central components and is easy to set up and maintain. It supports fault tolerance and malleability by implementing all functionality using P2P techniques. If resources used by an application are removed or fail, Zorilla can automatically find replacement resources. It was specifically designed to easily combine resources in multiple administrative domains.

To create a resource pool, a Zorilla daemon process must be started on each participating machine. Also, each machine must receive the address of at least one other machine to set up a connection. Jobs can be submitted to Zorilla using the JavaGAT or, alternatively, using a command-line interface. Zorilla then allocates the requested number of resources and schedules the application, taking user-defined requirements like memory size into account. The combination of virtualization and P2P techniques thus makes it very easy to deploy applications with Zorilla.

IV. Experimental Evaluation

To evaluate Ibis's functionality and performance, we carried out a series of experiments with the multimedia application. We used the Distributed ASCI Supercomputer 3 (DAS-3), a five-cluster distributed grid system in the Netherlands; additional clusters in Chicago, Japan, and Sydney the US East Amazon EC2 cloud system, and a desktop grid and single stand-alone machine, both in Amsterdam. Together, these machines comprised a real-world distributed system. We first compared the performance of Java/Ibis and C++/MPI implementations of the data-parallel processing of a single video frame. On a single machine the Java program is about 12 percent slower than the C++ version, which is within acceptable limits for a "compile once, run everywhere" application executing inside a virtual machine. On an 80-node DAS-3 cluster with the Myri-10G (Myricom 10-Gbit Ethernet) local network, the Java/Ibis and C++/MPI programs have very similar speedup (scalability) and communication overheads.

In the distributed version of our application, the data-parallel analysis is wrapped in a multimedia server. Client applications can upload an image or video frame to such a server and receive back a recognition result. When multiple servers are available, a client can use these simultaneously for task-parallel processing of subsequent images. We used to start a client on a local machine and to deploy four data-parallel multimedia servers, each on a different DAS-3 cluster (using 64 machines in total). All code was implemented in Java/Ibis, compiled on the client machine, and deployed directly from there. No application software or libraries were initially installed on any other machine. Using a single multimedia server resulted in a processing rate of approximately 1.2 frames per second. The simultaneous use of two and four clusters led to linear speedups at the client side of 2.5 and 5 frames per second, respectively. Adding additional clusters such as an EC2 cloud, a local desktop grid, and a local stand-alone machine improved the frame rate even further. We thereby obtained a worldwide system using a variety of grid middleware Globus, Zorilla, and SSH simultaneously from within a single application.

The Smart Sockets hub network circumvented a range of connectivity problems between the sites. Many sites have a firewall, and the Japan and Australia clusters can only be reached using SSH tunnels. In addition, almost all of the applied resources have more than one IP address. Smart Sockets automatically selected the appropriate addresses when two sites communicated. Without it, only the DAS-3, desktop grid, and stand-alone machine would have been reachable. To test Ibis's fault-tolerance mechanisms, we conducted an experiment in which an entire multimedia server crashed. The resource-tracking system noticed this crash and signaled the application. The client then removed the crashed server from the list of available servers. The application continued to run, with the client forwarding video frames to the remaining servers. We also accessed the multimedia servers from an HTC T-Mobile G1 smartphone, which used Ibis to upload pictures taken with the phone's camera and receive back a recognition result. Running the full application on the smartphone itself isn't possible due to CPU and memory limitations. Using the multimedia servers, however, the phone obtained a result in about three seconds. This clearly shows Ibis's potential to open up mobile computing to compute-intensive applications. Using Ibis Deploy, it's even possible to deploy the entire distributed application from the smartphone itself.

Fig 4:- The Ibis Deploy GUI lets users load applications and resources (top middle) and keep track of running processes (bottom half). The top left of the figure shows the locations of available resources; the top right shows the Smart Sockets network consisting of hubs and compute nodes.

V. OPEN PROBLEMS AND FUTURE WORK

The Ibis programming subsystem is mostly useful for applications written in Java or languages that compile to Java source code or byte code. Java applications can use non-Java libraries through the Java Native Interface and invoke non-Java executables with the Process. exec method. Theoretically, non-Java applications could also use the IPL through the JNI, but this is complicated. Despite these restrictions, many applications have been programmed on top of Ibis. In addition, the Ibis deployment subsystem has been used to deploy both Java and non-Java applications. We're currently researching how to integrate support for accelerators like GPUs, which requires access to non-Java code. In addition, existing high-level programming models don't cover all application domains, leaving some applications to use the IPL directly because they must address locality optimizations, fault tolerance, or malleability themselves. We're thus developing more flexible runtime support in Ibis for a broader range of high-level programming models. Finally, the interoperability layer (JavaGAT) introduces some runtime overhead. In practice, this overhead is insignificant except for operations that provide additional semantics such as remote error checks. More importantly, the JavaGAT intelligent-dispatching technique leads to more complex error reporting and debugging if operations fail. Instead of a single error message, the user now gets one error message per middleware layer that the JavaGAT attempted to use. Visual debugging and profiling tools should be developed to help the user address these problems.

Ibis drastically reduces the effort needed to create and deploy applications for real-world distributed systems that consist of ad hoc combinations of clusters, grids, clouds, desktop grids, stand-alone machines, and even mobile devices. To achieve this, it integrates solutions to many fundamental distributed computing problems in a single modular programming and deployment system, written entirely in Java. An important lesson learned from Ibis is that resource tracking functionality is as essential as communication functionality. While communication is among the basic capabilities of any distributed programming system, Ibis is one of the few systems that support resource tracking to implement fault tolerance and malleability. A second important lesson is that direct, two-way connectivity is rare in a real-world distributed system. However, SmartSockets achieves this in a transparent manner. Another lesson is that, for portability, it's not advisable to implement applications using one particular middleware system but to use a middleware-independent API, such as the JavaGAT, instead. Ibis also tries to make distributed programming easier by providing high-level programming models on top of these mechanisms. Satin, for example, makes fault tolerance and malleability transparent and automatically performs locality and latency-hiding optimizations.

REFERENCES

[1] Henri E. Bal, Jason Maassen, Rob V. van Nieuwpoort, Niels Drost, Roelof Kemp, Timo van Kessel, Nick Palmer, Gosia Wrzesiska,ThiloKielmann, Kees van Reeuwijk, Frank J. Seinstra, Ceriel J.H. Jacobs, and Kees Verstoep *Vrije **University. Amsterdam***

[2] H.E. Bal, N. Drost, R. Kemp, J. Maassen, R.V. van Nieuwpoort, C. van Reeuwijk, and F.J. Seinstra Department of Computer Science, Vrije Universiteit, De Boelelaan 1081A, 1081 HV Amsterdam, The Netherlands {bal, ndrost, rkemp, jason, rob, reeuwijk, fjseins}@cs.vu.nl

[3] Y. Yorozu, M. Hirano, K. Oka, and Y. Tagawa, "Electron spectroscopy studies on magneto-optical media and plastic substrate interface," IEEE Transl. J. Magn. Japan, vol. 2, pp. 740–741, August 1987 [Digests 9th Annual Conf. Magnetics Japan, p. 301, 1982].