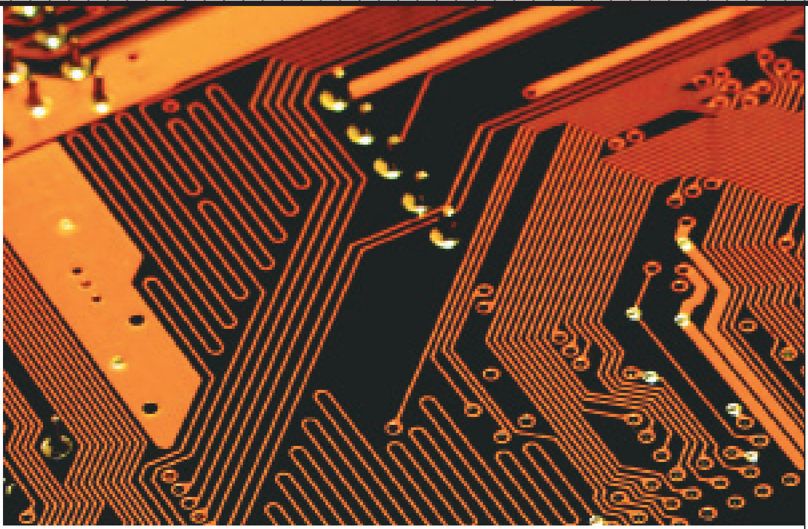
Real-World Distributed Computing with Ibis



Henri E. Bal, Jason Maassen, Rob V. van Nieuwpoort, Niels Drost, Roelof Kemp, Timo van Kessel, Nick Palmer, Gosia Wrzesińska, Thilo Kielmann, Kees van Reeuwijk, Frank J. Seinstra, Ceriel J.H. Jacobs, and Kees Verstoep
Vrije Universiteit, Amsterdam

The use of parallel and distributed computing systems is essential to meet the ever-increasing computational demands of many scientific and industrial applications. Ibis allows easy programming and deployment of compute-intensive distributed applications, even for dynamic, faulty, and heterogeneous environments.

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.[1]

Unfortunately, current practice shows that this goal remains out of reach.[2] For example, today's grid systems are mostly exploited to run coarse-grained parameter-sweep or master-worker programs. For more complex applications, grid usage is generally limited to straightforward scheduling systems that select a single site for execution. This is unfortunate, as many scientific and industrial applications—including astronomy, multimedia, medical imaging, and biobanking—would benefit from distributed compute resources. Optical networking advances also enable a much larger class of applications to run efficiently on such distributed systems.[3] In addition, research hasn't adequately addressed the problems that can arise from combining multiple unrelated systems to perform a single distributed computation. This is a likely scenario, as many scientific users have access to a wide variety of systems.

In itself, each cluster, grid, and cloud provides well-defined access policies, full connectivity, and middleware that allows 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, hardware, 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.

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*, as shown in Figure 1. Writing applications for such systems is notoriously difficult, as application programmers must take into account all of the problems described above. Deploying the applications is equally hard because each site is likely to have its own middleware and access policies.

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

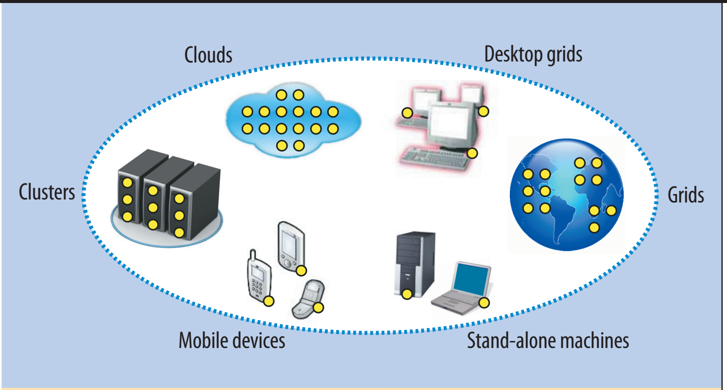


Figure 1. A "worst-case" real-world distributed system consisting of clusters, grids, and clouds as well as desktop grids, stand-alone machines, and mobile devices. Clusters, grids, and clouds are well-organized subsystems that use their own middleware, programming interfaces, access policies, and protection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

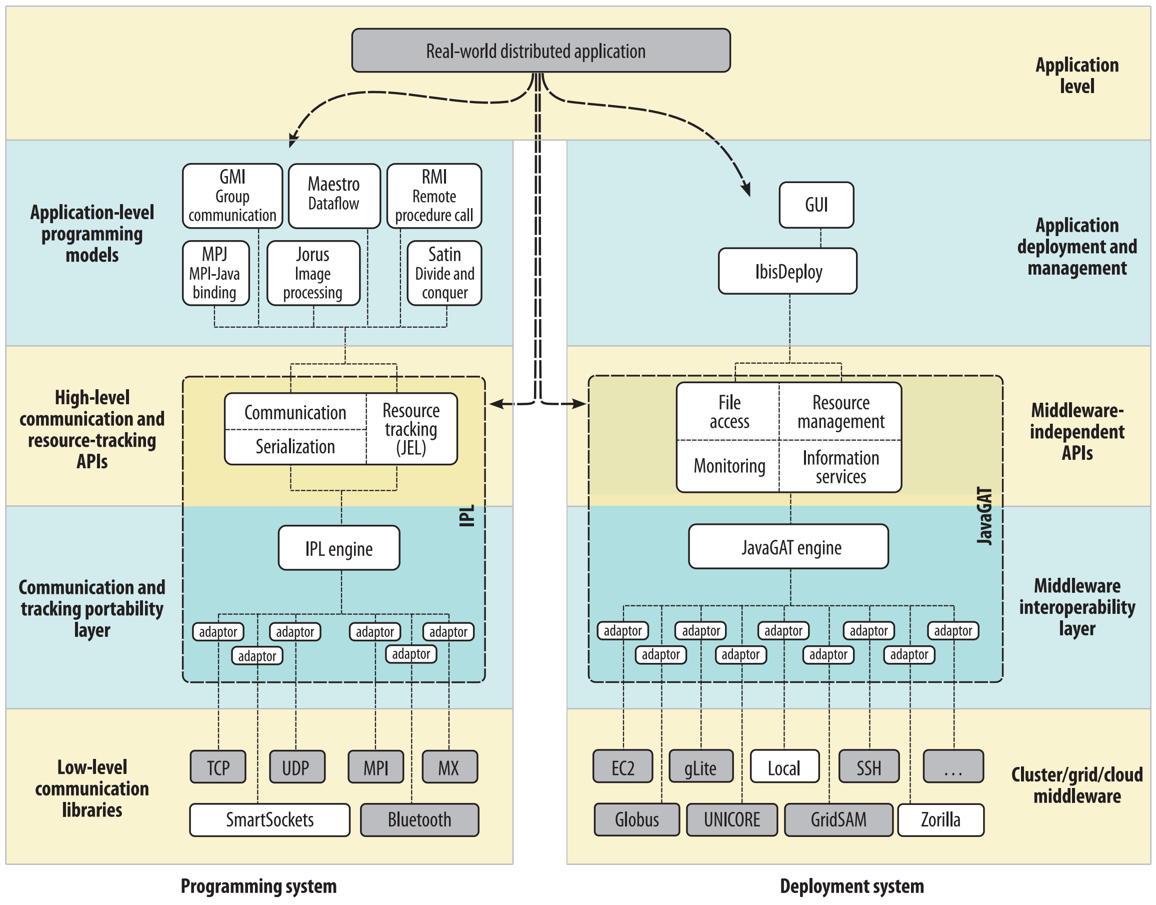


Figure 2. Ibis system architecture. The white boxes belong to Ibis; the gray ones represent third-party software.

RELATED WORK

Ibis provides a dual-subsystem solution for real-world distributed programming and deployment. In contrast, most existing systems focus on one of the two subsystems.[1]

ProActive,[2] like Ibis, follows a logical dual-subsystem approach. It contains several grid programming models and provides grid deployment and virtualization components. Whereas the Ibis IPL is small and highly efficient, the core of ProActive is more heavyweight. ProActive further requires the user to handle all connection setup problems and select the appropriate middleware.

Phoenix[3] consists of a general message-passing model that allows compute nodes to be added to and removed from a running application. It also deals with some of the connection setup problems solved by SmartSockets. Phoenix provides easy-to-use tools that handle common grid operations. However, it can't automatically exploit different grid middleware systems simultaneously.

The GRID superscalar framework[4] supports a certain degree of automatic grid deployment. It consists of an application programming interface, a runtime system, and a grid deployment center. It automatically converts a sequential application composed of tasks into a parallel application, allowing independent tasks to be executed on different grid resources.

The GridRPC specification[5] defines an API for remote procedure calls in grids. Two reference implementations exist, Ninf-G and NetSolve/GridSolve. In contrast to the JavaGAT, the binding of grid middleware to application objects is entirely static.

The Open Grid Forum is currently standardizing the next-generation grid programming toolkit: a Simple API for Grid Applications (SAGA).[6] The goal is to provide a simple, uniform, standard programming interface for distributed applications, with consistent semantics and style for different grid functionalities. Notably, SAGA's Java Reference Implementation is implemented directly on top of the JavaGAT.

References

1. I. Foster and C. Kesselman, eds., *The Grid 2: Blueprint of a New Computing Infrastructure*, 2nd ed., Morgan Kaufmann, 2003.
2. L. Baduel et al., "Programming, Deploying, Composing, for the Grid," *Grid Computing: Software Environments and Tools*, J.C. Cunha and O.F. Rana, eds., Springer, 2006, pp. 205-229.
3. K. Taura et al., "Phoenix: A Parallel Programming Model for Accommodating Dynamically Joining/Leaving Resources," *Proc. 9th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming* (PPoPP 03), 2003, ACM Press, pp. 216-229.
4. R. Badia et al., "Programming Grid Applications with GRID Superscalar," *J. Grid Computing*, June 2003, pp. 151-170.
5. K. Seymour et al., "Overview of GridRPC: A Remote Procedure Call API for Grid Computing," *Proc. 3rd Int'l Workshop Grid Computing*, LNCS 2536, Springer, 2002, pp. 274-278.
6. T. Goodale et al., "SAGA: A Simple API for Grid Applications, High-Level Application Programming on the Grid," *Computational Methods in Science and Technology*, vol. 12, no. 1, 2006, pp. 7-20.

where they run. It should also provide support for distributed file management, user authentication, resource management, and interoperability between different middleware systems.

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 users' needs.

IBIS ARCHITECTURE

Researchers are extensively studying high-performance and distributed computing tools and mechanisms in the field, but no existing single software system covers the full spectrum. Ibis aims to address this deficiency with an integrated system based on a straightforward, user-oriented philosophy: real-world distributed applications should be developed and compiled on a local workstation, and deployed from there onto the distributed system.

This "write-and-go" philosophy requires minimal assumptions about the execution environment. To enable applications to run in a heterogeneous system, Ibis exploits Java virtual machine technology. As Figure 2 shows, the Ibis system architecture follows the dual-subsystem approach. The programming system provides a range of programming models, all implemented on the same communication library: the Ibis Portability Layer (IPL). The deployment system contains a GUI and a library for deploying and managing applications, implemented on a middleware interoperability layer: the Java Grid Application Toolkit (JavaGAT).

Ibis is modular and flexible, allowing users to select the functionality they require from either Ibis or other software. For example, applications can use the high-level programming models and the deployment GUI, but they can also be implemented directly on the IPL and JavaGAT. Likewise, applications are free to use only one of the two subsystems. For example, the deployment system can be used to deploy C/MPI and other non-Java applications.

Ibis is fully open source and used in various real-world distributed applications such as multimedia computing, spectroscopic data processing (FOM Institute for Atomic and Molecular Physics), human brain-scan analysis (Vrije Universiteit Medical Center), automatic grammar learning, and many others. In addition, Ibis has been used to build high-level programming systems, including a workflow engine for astronomy applications in D-Grid (Max Planck Institute for Astrophysics), the GridChem gateway for TeraGrid, and a grid file system (University of Erlangen-Nürnberg).

Ibis has also been applied to enhance existing systems such as ProActive (INRIA), Jylab (University of Patras), and the GRID superscalar framework (Barcelona Supercomputer Center). The "Related Work" sidebar describes some of these systems and how they differ from Ibis.

Ibis has won prizes in numerous international competitions including the International Scalable Computing

Challenge at CCGrid 2008 (for scalability), the International Data Analysis Challenge for Finding Supernovae at IEEE Cluster/Grid 2008 (for speed and fault tolerance), and the Billion Triples Challenge at the 2008 International Semantic Web Conference (for general innovation).

IBIS PROGRAMMING SYSTEM

The Ibis programming system provides many programming models, all implemented on top of the IPL.

IPL

The IPL is a Java-based communication library that provides robust communication and resource-tracking mechanisms. It typically ships with the application as jar (Java archive) files, so no additional preinstalled libraries need be present at any destination machine.

The library provides a range of communication primitives including those for point-to-point and multicast communication. It supports streaming communication, which is especially important in high-latency environments as this allows overlapping of serialization, communication, and deserialization of data. The IPL avoids copying overhead as much as possible and uses bytecode rewriting to generate efficient serialization and deserialization functions. Consequently, it can significantly outperform Sun remote method invocation (RMI) communication and even C/MPI, in particular for complex data structures.[4]

The IPL has been designed specifically for real-world distributed environments where resources can be added or removed dynamically. It incorporates a mechanism, Join-Elect-Leave (JEL), that tracks which resources are being used and what roles they have. JEL is based on the concept of signaling: it notifies the application or runtime system when resources are added to or removed from the computation. To select resources with a special role, it includes elections. JEL thus provides the building blocks for fault tolerance and malleability by giving an up-to-date view of available resources, allowing applications and runtime systems to respond to changes when required.

A number of pure-Java IPL implementations are available using the Ibis SmartSockets library, TCP (Transmission Control Protocol), UDP (User Datagram Protocol), and Bluetooth. In addition, we provide implementations using specialized non-Java libraries, such as MX (Myrinet) or MPI. The SmartSockets, TCP, and UDP implementations also work on the Android smartphone platform.

SmartSockets

Running a parallel application on distributed resources is complicated due to connectivity problems that make direct communication difficult or impossible. Incoming traffic at a node may be restricted by a firewall or NAT. The presence of multiple network interfaces and IP addresses can cause addressing problems, as can private network addresses. The SmartSockets library automatically solves such problems using existing and novel solutions, including reverse connection setup, overlay routing, and Secure Shell (SSH) tunneling.

SmartSockets creates an overlay network with a set of interconnected support processes called *hubs*. Employing gossiping techniques, the hubs discover which other hubs they can connect to, using SSH tunneling if necessary. This overlay network can be used to help solve connectivity problems or route the application's network traffic.

When creating a connection, SmartSockets initially tries to set up a regular (direct) TCP connection. If this fails, it uses the overlay network to send a request for a reverse connection setup to the target machine. If this also fails, the library creates a virtual connection that uses overlay routing. SmartSockets also handles machines with multiple network addresses via multihoming. During connection setup, it considers all source and target addresses. It uses heuristics to determine the combinations most likely to work and then tries these first. Extra identity checks in the procotol ensure that it reaches the correct target machine. A worldwide experiment[5] demonstrated the library's effectiveness: in 30 realistic connectivity scenarios, SmartSockets was always capable of establishing a connection, while traditional sockets only worked in six of these.

The IPL has been designed specifically for real-world distributed environments where resources can be added or removed dynamically.

Ibis programming models

Besides the IPL, Ibis provides a range of programming models, from low-level message passing to high-level divide-and-conquer parallelism. It implements the following programming models using the IPL:

- MPJ, the MPI binding for Java;
- Satin, a divide-and-conquer model;[6]
- RMI, object-oriented remote procedure call;
- group method invocation (GMI), a generalization of RMI to group communication, including multicast and all-to-all communication;
- Maestro, a fault-tolerant and self-optimizing dataflow model;[7] and
- Jorus, a programming model for data-parallel multimedia applications.[8]

The higher-level programming models' runtime systems exploit the IPL and JEL to address many distributed programming difficulties. A good example is Satin, which uses JEL resource-tracking mechanisms to make applications malleable and fault tolerant. For example, Satin can reexecute subtasks if a processor crashes. Also, it can dynamically schedule subtasks on new machines that become available during the computation, and it can migrate subtasks if machines leave the computation. Satin's scheduler also does locality optimizations: divide-and-conquer programs are inherently hierarchical and can therefore be mapped efficiently onto a hierarchical wide-area system such as a grid. Likewise, the scheduler does latency hiding; if it needs to retrieve new jobs from distant machines, it will do that asynchronously, without blocking.[6]

The JavaGAT uses intelligent dispatching to integrate multiple middleware systems with different and incomplete functionality into a single, consistent system.

Satin thus makes these problems transparent to the user and application. Applications written with a lower-level programming model like the IPL must deal with such problems explicitly, but Ibis gives them the necessary low-level mechanisms to do so.

IBIS DEPLOYMENT SYSTEM

The Ibis deployment system consists of a software stack and a graphical user interface for deploying and managing applications. The GUI is implemented on top of the JavaGAT.

JavaGAT

Writing distributed applications using existing middleware APIs is a daunting task. APIs change frequently and are often incomplete and too low-level.[9] The JavaGAT provides a high-level API that facilitates development of complex applications. This API is object oriented and offers high-level primitives for access to the distributed system, independent of the middleware that implements this functionality. The primitives provide access to remote data, start remote jobs, and support monitoring, steering, user authentication, resource management, and storing of application-specific data. The JavaGAT uses an extensible architecture, wherein *adaptors* (plugins) provide access to the different types of middleware.

The JavaGAT also uses *intelligent dispatching* to integrate multiple middleware systems with different and incomplete functionality into a single, consistent system. This technique dynamically forwards application calls on the JavaGAT API to one or more middleware adaptors that implement the requested functionality. Selection occurs at runtime and uses policies and heuristics that automatically select the best available middleware, enhancing portability. If an operation fails, the intelligent-dispatching feature will automatically select and dispatch the API call to an alternative middleware. This process continues until a middleware successfully performs the requested operation.

Although such flexibility comes at the cost of some runtime overhead, this is often negligible compared to the cost of the operations themselves. For instance, a Globus job submission takes several seconds, while the overhead introduced by the JavaGAT is less than 10 ms. However, the additional semantics of the high-level API can introduce some overhead. For instance, if a file is copied, the JavaGAT first checks if the destination already exists or is a directory. These extra checks cost time because they require remote operations.

The JavaGAT API isn't the lowest common denominator of the underlying middleware APIs. Instead, the JavaGAT offers rich default functionality and can combine features of multiple middleware layers with its intelligent-dispatching technique. Consider the following real code example, which copies remote files and entire directories between sites:

```java
import org.gridlab.gat.*;
import org.gridlab.gat.io.File;

public class Copy {
  public static void main(String [] args)
  throws Exception {
    GATContext context = new
    GATContext();
    URI source = new URI(args[0]);
    URI dest = new URI(args[1]);

    File file = GAT.createFile(context,
    source); // Create a GAT file

    file.copy(dest); // The actual file
    or directory copy

    GAT.end(); // Shutdown the JavaGAT
  }
}
```

This code is middleware independent and demonstrates the JavaGAT API's power; equivalent Globus code would take hundreds of lines. The JavaGAT allows programmers to ignore system-level peculiarities and instead focus on domain-specific problems.

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. Currently, the JavaGAT supports many different middleware systems such as Globus, UNICORE (Uniform Interface to Computing Resources), gLite, PBS (Portable Batch System), SGE (Sun Grid Engine), KOALA (a co-allocating grid scheduler), SSH, GridSAM, Amazon EC2 (Elastic Compute Cloud), ProActive, GridFTP, HTTP, SMB (Server Message Block)/CIFS (Common Internet File System), and Zorilla.

Zorilla

Most existing middleware APIs lack coscheduling 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.

IbisDeploy

Many applications use the same deployment process. Therefore, IbisDeploy 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, IbisDeploy starts the SmartSockets hub network automatically. It also automatically uploads the program codes, libraries, and input files (*prestaging*) and automatically downloads the output files (*poststaging*).

The IbisDeploy 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.

APPLICATION EXAMPLE: MULTIMEDIA CONTENT ANALYSIS

We illustrate Ibis with an application that performs real-time recognition of everyday objects. Images produced by a camera are processed by an advanced algorithm that extracts *feature vectors* from the video data, which describe local properties like color and shape. To recognize an object, the application compares the object's feature vectors to ones stored earlier and annotated with a name.

IbisDeploy provides a simple and generic API and GUI that can automatically perform commonly used deployment scenarios.

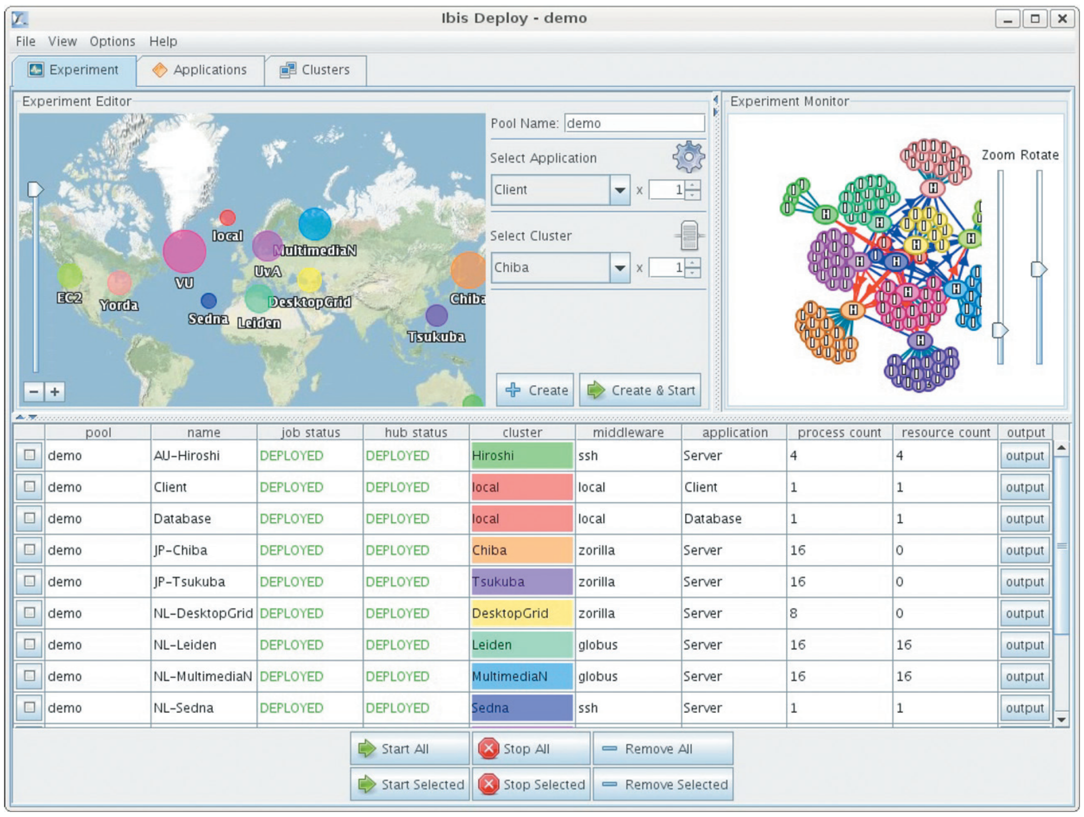
As this is a compute-intensive problem with soft real-time constraints, a large distributed system performs the analysis.[8] A data-parallel application running on a single site processes a single video frame. Calculations over consecutive frames are distributed over different sites in a task-parallel manner.

The initial application was written in C++/MPI, using TCP and SSH tunnels for wide-area communication. This program used manual deployment, was vulnerable to connectivity problems and partial failures (each causing the entire application to fail), and was frustrating to write and maintain on heterogeneous hardware and middleware. Step by step, we replaced all its components with an implementation in Java and Ibis. With the new program, the IbisDeploy GUI makes deployment easy, SmartSockets automatically corrects the connectivity problems, and the JavaGAT accommodates the middleware heterogeneity. We provided robustness by adding fault tolerance and malleability support to the application using the IPL-provided mechanisms.

The resulting application is compiled on a desktop machine and easily deployed onto a distributed system. It concurrently uses up to 20 clusters, commonly employing a total of 500 to 800 cores, and a mix of different middleware. The application even runs on the Android smartphone platform, allowing distributed object recognition from mobile devices.

EXPERIMENTAL EVALUATION

To evaluate Ibis's functionality and performance, we carried out a series of experiments with the multimedia



Pool Name: demo

Select Application: Client x 1

Select Cluster: Chiba x 1

	pool	name	job status	hub status	cluster	middleware	application	process count	resource count	output
☐	demo	AU-Hiroshi	DEPLOYED	DEPLOYED	Hiroshi	ssh	Server	4	4	output
☐	demo	Client	DEPLOYED	DEPLOYED	local	local	Client	1	1	output
☐	demo	Database	DEPLOYED	DEPLOYED	local	local	Database	1	1	output
☐	demo	JP-Chiba	DEPLOYED	DEPLOYED	Chiba	zorilla	Server	16	0	output
☐	demo	JP-Tsukuba	DEPLOYED	DEPLOYED	Tsukuba	zorilla	Server	16	0	output
☐	demo	NL-DesktopGrid	DEPLOYED	DEPLOYED	DesktopGrid	zorilla	Server	8	0	output
☐	demo	NL-Leiden	DEPLOYED	DEPLOYED	Leiden	globus	Server	16	16	output
☐	demo	NL-MultimediaN	DEPLOYED	DEPLOYED	MultimediaN	globus	Server	16	16	output
☐	demo	NL-Sedna	DEPLOYED	DEPLOYED	Sedna	ssh	Server	1	1	output

Figure 3. The IbisDeploy 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 SmartSockets network consisting of hubs and compute nodes. A video presentation is available at www.cs.vu.nl/ibis/demos.html.

application (see www.cs.vu.nl/ibis/demos.html for a video demonstration). We used the Distributed ASCI Supercomputer 3 (DAS-3), a five-cluster distributed grid system in the Netherlands; additional clusters in Chicago, Japan (the Chiba and Tsukuba InTrigger sites), 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 IbisDeploy 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 SmartSockets 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.

SmartSockets 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 IbisDeploy, it's even possible to deploy the entire distributed application from the smartphone itself.

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 bytecode. 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's 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.

Ibis can be downloaded for free at www.cs.vu.nl/ibis.

References

1. I. Foster, C. Kesselman, and S. Tuecke, "The Anatomy of the Grid: Enabling Scalable Virtual Organizations," *Int'l J. High-Performance Computing Applications*, Aug. 2001, pp. 200-222.
2. D. Butler, "The Petaflop Challenge," *Nature*, 5 July 2007, pp. 6-7.
3. K. Verstoep et al., "Experiences with Fine-Grained Distributed Supercomputing on a 10G Testbed," *Proc. 2008 8th IEEE Int'l Symp. Cluster Computing and the Grid* (CCGrid 08), IEEE CS Press, 2008, pp. 376-383.
4. R.V. van Nieuwpoort et al., "Ibis: A Flexible and Efficient Java-Based Grid Programming Environment," *Concurrency and Computation: Practice and Experience*, June 2005, pp. 1079-1107.
5. J. Maassen and H.E. Bal, "SmartSockets: Solving the Connectivity Problems in Grid Computing," *Proc. 16th Int'l Symp. High Performance Distributed Computing* (HPDC 07), ACM Press, 2007, pp. 1-10.
6. R.V. van Nieuwpoort et al., "Satin: A High-Level and Efficient Grid Programming Model," *ACM Trans. Programming Languages and Systems*, Mar. 2010, article no. 9.
7. C. van Reeuwijk, "Maestro: A Self-Organizing Peer-to-Peer Dataflow Framework Using Reinforcement Learning," *Proc. 18th ACM Int'l Symp. High Performance Distributed Computing* (HPDC 09), ACM Press, 2009, pp. 187-196.
8. F.J. Seinstra et al., "High-Performance Distributed Video Content Analysis with Parallel-Horus," *IEEE MultiMedia*, Oct. 2007, pp. 64-75.

9. R. Medeiros et al., “Faults in Grids: Why Are They So Bad and What Can Be Done About It?” *Proc. 4th Int’l Workshop Grid Computing* (GRID 03), IEEE CS Press, 2003, pp. 18-24.
10. R.V. van Nieuwpoort, T. Kielmann, and H.E. Bal, “User-Friendly and Reliable Grid Computing Based on Imperfect Middleware,” *Proc. 2007 ACM/IEEE Conf. Supercomputing* (SC 07), ACM Press, 2007, article no. 34.

***Henri E. Bal** is a full professor in the Department of Computer Science, where he heads the High Performance Distributed Systems research group, at Vrije Universiteit, Amsterdam, the Netherlands. Contact him at bal@cs.vu.nl.*

***Jason Maassen** is a postdoctoral researcher in the Department of Computer Science at Vrije Universiteit and one of the original designers of the IPL and SmartSockets. Contact him at jason@cs.vu.nl.*

***Rob V. van Nieuwpoort** is a postdoctoral researcher in the Department of Computer Science at Vrije Universiteit and ASTRON (Netherlands Institute for Radio Astronomy) and one of the original designers of the IPL and the JavaGAT. Contact him at rob@cs.vu.nl.*

***Niels Drost** is a postdoctoral researcher in the Department of Computer Science at Vrije Universiteit and the original designer of Zorilla. Contact him at niels@cs.vu.nl.*

***Roelof Kemp** is a PhD student in the Department of Computer Science at Vrije Universiteit. Contact him at rkemp@cs.vu.nl.*

***Timo van Kessel** is a PhD student in the Department of Computer Science at Vrije Universiteit. Contact him at timo@cs.vu.nl.*

***Nick Palmer** is a PhD student in the Department of Computer Science at Vrije Universiteit. Contact him at palmer@cs.vu.nl.*

***Gosia Wrzesinska** is a senior software engineer at VectorWise, where she works on high-performance query processing for database engines, and received a PhD in computer science from Vrije Universiteit. Contact her at gosia@vectorwise.com.*

***Thilo Kielmann** is an associate professor in the Department of Computer Science at Vrije Universiteit and a steering group member of the Open Grid Forum and Gridforum Nederland. Contact him at kielmann@cs.vu.nl.*

***Kees van Reeuwijk** is a postdoctoral researcher in the Department of Computer Science at Vrije Universiteit. Contact him at reeuwijk@cs.vu.nl.*

***Frank J. Seinstra** is an assistant professor in the Department of Computer Science at Vrije Universiteit. Contact him at fjseins@cs.vu.nl.*

***Ceriel J.H. Jacobs** is a scientific programmer in the Department of Computer Science at Vrije Universiteit and the maintainer of the Ibis software. Contact him at ceriel@cs.vu.nl.*

***Kees Verstoep** is a scientific programmer in the Department of Computer Science at Vrije Universiteit. Contact him at versto@cs.vu.nl.*