European Research Network on Foundations, Software Infrastructures and Applications for large scale distributed, GRID and Peer-to-Peer Technologies
A Network of Excellence funded by the European Commission

Grid Application Programming Environments

Thilo Kielmann, Andre Merzky, Henri Bal
Vrije Universiteit
Amsterdam, The Netherlands
{kielmann,merzky,bal}@cs.vu.nl
Francoise Baude, Denis Caromel, Fabrice Huet
INRIA, I3S-CNRS, UNSA
Sophia Antipolis France
{fbaude,dcaromel,fhuet}@sophia.inria.fr

CoreGRID Technical Report
Number TR-0003
June 21, 2005

Institute on Problem Solving Environment, Tools and GRID Systems

CoreGRID - Network of Excellence
URL: http://www.coregrid.net

CoreGRID is a Network of Excellence funded by the European Commission under the Sixth Framework Programme

Project no. FP6-004265

Grid Application Programming Environments

Thilo Kielmann, Andre Merzky, Henri Bal
Vrije Universiteit
Amsterdam, The Netherlands
{kielmann,merzky,bal}@cs.vu.nl
Francoise Baude, Denis Caromel, Fabrice Huet
INRIA, I3S-CNRS, UNSA
Sophia Antipolis France
{fbaude,dcaromel,fhuet}@sophia.inria.fr

CoreGRID TR-0003
June 21, 2005

Abstract

One challenge of building future grid systems is to provide suitable application programming interfaces and environments. In this chapter, we identify functional and non-functional properties for such environments. We then review three existing systems that have been co-developed by the authors with respect to the identified properties: ProActive, Ibis, and GAT. Apparently, no currently existing system is able to address all properties. However, from our systems, we can derive a generic architecture model for grid application programming environments, suitable for building future systems that will be able to address all the properties and challenges identified.

This research work is carried out under the FP6 Network of Excellence CoreGRID funded by the European Commission (Contract IST-2002-004265).

1 Introduction

A grid, based on current technology, can be considered as a distributed system for which heterogeneity, wide-area distribution, security and trust requirements, failure probability, as well as high latency and low bandwidth of communication links are exacerbated. Different grid middleware systems have been built, such as the Globus toolkit [39], EGEE LCG and g-lite [12], ARC [35], Condor [20], Unicore [18], etc). All these systems provide similar grid services, and a convergence is in progress. As the GGF [22] definition of grid services tries to become compliant to Web services technologies, a planetary-scale grid system may emerge, although this is not yet the case. We thus consider a grid a federation of different heterogeneous systems, rather than a virtually homogeneous distributed system.

In order to build and program applications for such federations of systems, (and likewise application frameworks such as problem solving environments or "virtual labs"), there is a strong need for solid high-level middleware, directly interfacing application codes. Equivalently, we may call such middleware a grid programming environment. Indeed, grid applications require the middleware to provide them with access to services and resources, in some simple way. Accordingly, the middleware should implement this access in a way that hides heterogeneity, failures, and performance of the federation of resources and associated lower-level services they may offer. The challenge for the middleware is to provide applications with APIs that make applications more or less grid unaware (i.e. the grid becomes invisible).

Having several years of experience designing and building such middleware, we analyze our systems, aiming at a generalization of their APIs and architecture that will finally make them suitable for addressing the challenges and properties of future grid application programming environments. In Section 2, we identify functional and non-functional properties for future grid programming environments. In Section 3, we present our systems, ProActive, Ibis, and GAT, and investigate which of the properties they meet already. In Section 4, we discuss related systems by other authors. Section 5 then derives a generalized architecture for future grid programming environments. In Section 6 we draw our conclusions and outline directions of future work.

2 Properties for grid application programming environments

Grid application programming environments provide both application programming interfaces (APIs) and runtime environments implementing these interfaces, allowing application codes to run in a grid environment. In this section, we outline the properties of such programming environments.

2.1 Non-functional properties

We begin our discussion with the non-functional properties as these are determining the constraints on grid API functionality. As such, issues like performance, security, and fault-tolerance have to be taken into account when designing grid application programming environments.

Performance

As high-performance computing is one of the driving forces behind grids, performance is the most prominent, non-functional property of the operations that implement the functional properties as outlined below. Job scheduling and placement is mostly driven by expected execution times, while file access performance is strongly determined by bandwidth and latency of the network, and the choice of the data transfer protocol and its configuration (like parallel TCP streams [34] or GridFTP [2]). The trade-off between abstract functionality and controllable performance is a classic since the early days of parallel programming [7]. In grids, it even gains importance due to the large physical distances between the sites of a grid.

Fault tolerance

Most operations of a grid API involve communication with physically remote peers, services, and resources. Because of this remoteness, the instabilities of network (Internet) communication, the fact that sites may fail or become unreacheable, and the administrative site autonomy, various error conditions arise. (Transient) errors are common rather

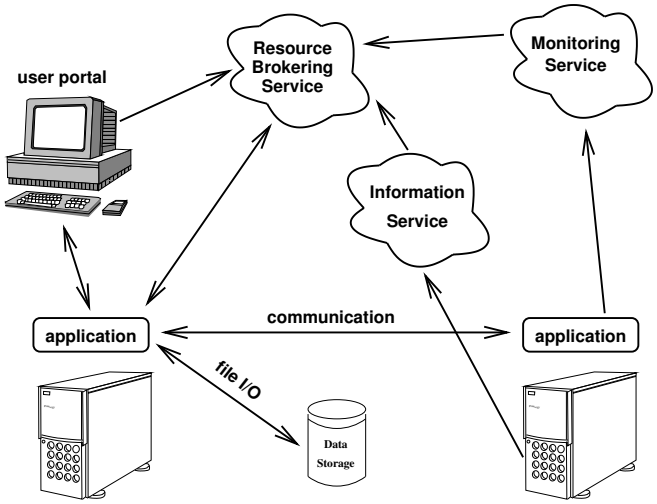

Figure 1: Grid application execution scenario

than the exception. Consequently, error handling becomes an integral part, both of grid runtime environments and of grid APIs.

Security and trust

Grids integrate users and services from various sites. Communication is typically performed across insecure connections of the Internet. Both properties require mechanisms for ensuring security of and trust among partners. A grid API thus needs to support mutual authentication of users and resources. Access control to resources (authorization) becomes another source of transient errors that runtime systems and their APIs have to handle. Besides authentication and authorization, privacy becomes important in Internet-based systems which can be ensured using encryption. Whereas encryption need not be reflected in grid APIs themselves, users may notice its presence by degraded communication performance.

Platform independence

It is an important property for programming environments to keep the application code independent from details of the grid platform, like machine names or file system layouts for application executables and data files. This needs to be reflected in the APIs provided by a grid programming environment. The corresponding implementations need to map abstract, application-level resources to their physical counterparts.

2.2 Functional properties

Figure 1 illustrates the involvement of programming environments in application execution scenarios. We envision the following categories of necessary functionality.

Access to compute resources, job spawning and scheduling

Users enter application jobs to the grid via some form of job submission tool, like *globusrun* [39], or a portal like GridSphere [32]. In simple cases, a job will run on a single resource or site. In more advanced scenarios, like dynamic grid applications [3] or in the general case of task-flow applications [5, 37], a running job will spawn off further jobs to available grid resources. But even the portal can be seen as a specialized grid application that needs to submit jobs.

A job submission API has to take descriptions of the job and of suitable compute resources. Only in the simplest cases will the runtime environment have (hard coded) information about job types and available machines. In any real-world grid environment, the mapping and scheduling decision is taken by an external resource broker service [33]. Such an external resource broker is able to take dynamic information about resource availability and performance into account.

Access to file and data resources

Any real-world application has to process some form of input data, be it files, data bases, or streams generated by devices like radio telescopes [38] or the LHC particle collider [23]. A special case of input files is the provisioning of program executable files to the sites on which a job has been scheduled. Similarly, generated output data has to be stored on behalf of the users.

As grid schedulers place jobs on computationally suitable machines, data access immediately becomes remote. Consequently, a grid file API needs to abstract from physical file locations while providing a file-like API to the data ("open, read/write, close"). It is the task of the runtime environment to bridge the gap between seemingly local operations and the remotely stored data files.

Communication between parallel and distributed processes

Besides access to data files, the processes of a parallel application need to communicate with each other to perform their tasks. Several programming models for grid applications have been considered in the past, among which are MPI [25, 26, 27], shared objects [28], or remote procedure calls [11, 36]. Besides suitable programming abstractions, grid APIs for inter-process communication have to take the properties of grids into account, like dynamic (transient) availability of resources, heterogeneous machines, shared networks with high latency and bandwidth fluctuations. The trade-off between abstract functionality and controllable performance is the crux of designing communication mechanisms for grid applications. Besides, achieving mere connectivity is a challenging task for grid runtime environments, esp. in the presence of firewalls, local addressing schemes, and non-IP local networks [17].

Application monitoring and steering

In case the considered grid applications are intended to be long running, users need to be in control of their progress in order to avoid costly repetition of unsuccessful jobs. For this purpose, users need to inspect and possibly modify the status of their application while it is running on some nodes in a grid. For this purpose, monitoring and steering interfaces have to be provided, such that users can interact with their applications. For this purpose, additional communication between the application and external tools like portals or application managers are required.

As listed so far, we consider these properties as the direct needs of grid application programs. Further functionality, like resource lookup, multi-domain information management, or accounting are of equal importance. However, we consider such functionality to be of indirect need only, namely within auxiliary grid services rather than the application programs themselves.

3 Existing grid programming environments

After having identified both functional and non-functional properties for grid application programming environments, we now present three existing systems, developed by the authors and their colleagues. For each of them, we outline their purpose and intended functionality, and we discuss which of the non-functional properties can be met. For the three systems, ProActive, Ibis, and GAT, we also outline their architecture and implementation.

3.1 ProActive

ProActive is a Java library for parallel, distributed and concurrent computing, also featuring mobility and security in a uniform framework. With a reduced set of simple primitives, ProActive provides a comprehensive API masking

the specific underlying tools and protocols used, and allowing to simplify the programming of applications that are distributed on a LAN, on a cluster of PCs, or on Internet Grids. The library is based on an active object pattern, on top of which a component-oriented view is provided.

Architecture All active objects are running in a JVM and more precisely are attached to a *Node* hosted by it. Nodes and active objects on the same JVM are indeed managed by a ProActive runtime (see Figure 2) which provides them support/services, such as lookup and discovery mechanism for nodes and active objects, creation of runtime on remote hosts, enactment of the communications according to the chosen transport protocol, security policies negociation, etc. Numerous meta-level objects are attached to an active object in order to implement features like remote communication, migration, groups, security, fault-tolerance and components. A ProActive runtime inter operates with an open and moreover extensible palette of protocols and tools; for communication and registry/discovery, security: RMI, Ibis, Jini (for environment discovery), web service exportation, HTTP, RMI over ssh tunneling; for process (JVM) deployment: ssh, sshGSI, rsh, Globus (through the JavaCog Kit API), LSF, PBS, Sun Grid Engine. Standard Java dynamic class loading is considered as the means to solve provision of code.

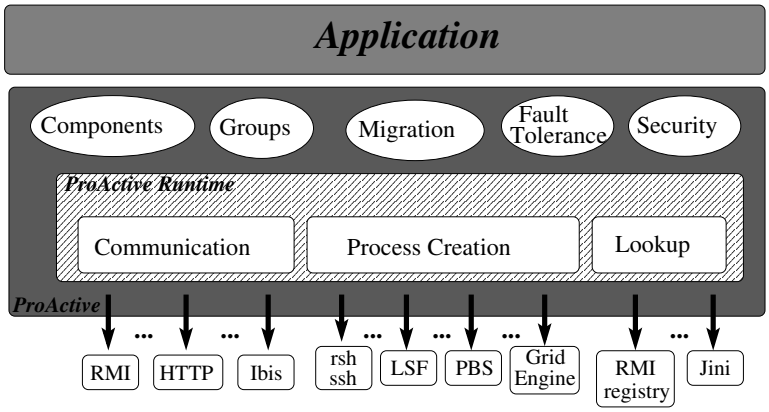

Figure 2: Application running on top of the ProActive architecture

3.1.1 Functional properties of ProActive

Access to computing resources, job spawning and scheduling Instead of having an API which mandates the application to configure the infrastructure it must be deployed on, deployment descriptors are used. They collect all the deployment information required when launching or acquiring already running ProActive runtimes (e.g. remote login information, CLASSPATH value) and ProActive proposes the notion of a Virtual Node which serves to virtualize the active object's location in the source code (see figure 3). Besides, the mapping from a virtual node to effective JVMs is managed via these deployment descriptors [9]. In summary, ProActive provides an open deployment system (through a minimum size API, associated with XML deployment files) that enables to access and launch JVMs on remote computing resources using an extensible palette of access protocols. ProActive provides a simple API to trigger (weak) migration of active objects as a means to dynamically remap activities on the target infrastructure (e.g. `ProActive.migrateTo(Node anode)`).

Communication for parallel processes Method calls sent to active objects are always asynchronous with transparent *future objects* and synchronization is handled by a mechanism known as *wait-by-necessity* [14]. Each active object has its own thread of control, and decides in which order to serve the incoming method call requests. Based on a simple Meta-Object Protocol, a communication between those active objects follows the standard Java method invocation syntax. Thus no specific API is required in order to let them communicate. ProActive provides an extension of this to groups of active objects, as a *typed group communication* mechanism. On a Java class, named e.g. `A`, here is an example of a typical group creation and method call

```
// A group of type "A" and its 3 members are created at once on the nodes
```

```
ProActiveDescriptor pad = ProActive.getProActiveDescriptor(String xmlFileLocation);
//---- Returns a ProActiveDescriptor object from the xml file
VirtualNode dispatcher = pad.getVirtualNode("Dispatcher");
//---- Returns the VirtualNode Dispatcher described in the xml file as an object
dispatcher.activate()
// --- Activates the VirtualNode
Node node = dispatcher.getNode();
//-----Returns the first node available among nodes mapped to the VirtualNode
C3DDispatcher c3dDispatcher = ProActive.newActive("org.objectweb.proactive.core.
examples.c3d.C3DDispatcher", param, node);
//-----Creates an active object running class C3DDispatcher, on the remote JVM.
```

Figure 3: Example of a ProActive source code for descriptor-based mapping

```
// directly specified, parameters are specified in params,
Object[][] params = {{...}, {...}, {...}};
A ag = (A) ProActiveGroup.newGroup("A",params, {node1,node2,node3});
V vg=ag.foo(pg); // A typed group communication
```

The *result* of a typed group communication, which may also be a group, is transparently built at invocation time, with a future for each elementary reply. It will be dynamically updated with the incoming results. Other features are provided through methods of the group API: parameter dispatching instead of broadcasting, using *scatter* groups, explicit group method call synchronization through futures (e.g. `waitOne`, `waitAll`), dynamic group manipulation by first getting the group representation, then `add`, `remove` of members. Groups provide an object oriented SPMD programming model [6].

Application monitoring and steering ProActive applications can be monitored transparently by an external application, written in ProActive: IC2D (Interactive Control and Debugging for Distribution) [9]. Once launched, it is possible to select some hosts (and implicitly all corresponding virtual nodes, nodes, and hosted active objects): this triggers the registration of a listener of all sort of events that occur (send of method calls, reception of replies, waiting state); they are sent to IC2D, then graphically presented. IC2D provides a drag-and-drop migration of active objects from one JVM to an other, which can be considered as a steering operation. A job abstraction enables to consider at once the whole set of activities and JVMs that correspond to the specific execution of an application. Graphical job monitoring is then provided, e.g. to properly kill it.

3.1.2 Non-functional properties of ProActive

Performance As a design choice, communication is asynchronous with futures. Compared to traditional futures, (1) they are created implicitly and systematically, (2) and can be passed as parameters to other active objects. As such, performance may come from a good overlapping of computation and communication. If pure performance is a concern, then ProActive should preferably use Ibis instead of standard RMI, as the transport protocol [24].

Fault tolerance Non functional exceptions are defined and may be triggered, for each sort of feature that may fail due to distribution. Handlers are provided to transparently manage those non-functional exceptions, giving the programmer the ability to specialize them. Besides, a ProActive application can transparently be made fault-tolerant. On user demand, a transparent checkpointing protocol, designed in relation with an associated recovery protocol, can be applied. Once any active object is considered failed, the whole application is restarted from a coherent global state. The only user's involvement is to indicate in the deployment descriptor, the location of a stable storage for checkpoints.

Security and trust The ProActive security framework allows to configure security according to the deployment of the application. Security mechanisms apply to basic features like communication authentication, integrity, confidentiality to more high-level ones like object creation or migration (e.g. a node may accept or deny the migration of an active object according to its provenance). Security policies are expressed outside the application, in a security descriptor attached to the deployment descriptor that the application will use. Policies are negotiated dynamically by the participants involved in a communication, a migration, or a creation, be they active objects or nodes, and according

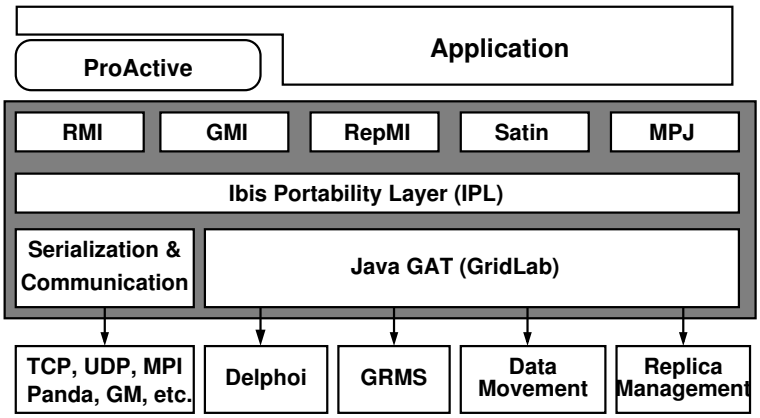

Figure 4: API's and architecture of the Ibis system

to their respective IP domain and ProActive runtime they are hosted by. The security framework is based on PKI. It is possible to tunnel over SSH all communications towards a ProActive runtime, as such multiplexing and demultiplexing all RMI connections or HTTP class loading requests to that host through its ssh port. For users, it requires only to specify in ProActive deployment descriptors, for instance, which JVMs should export their RMI objects through a SSH tunnel.

Platform independence ProActive is built in such a way as it does not require any modification to the standard Java execution environment, nor does it make use of a special compiler, pre-processor or modified virtual machine.

3.2 Ibis

The Ibis Grid programming environment [42] has been developed to provide parallel applications with highly efficient communication API's. Ibis is based on the Java programming language and environment, using the "write once, run anywhere" property of Java to achieve portability across a wide range of Grid platforms. Ibis aims at Grid-unaware applications. As such, it provides rather high-level communication API's that hide Grid properties and fit into Java's object model.

The Ibis runtime system architecture is shown in Figure 4. Ibis can be configured dynamically at run time, allowing to combine standard techniques that work "anywhere" (e.g., using TCP) with highly-optimized solutions that are tailored for special cases, like a local Myrinet interconnect. The *Ibis Portability Layer* (IPL), that provides this flexibility, consists of a small set of well-defined interfaces. The IPL can have different implementations, which can be selected and loaded into the application *at run time*.

The IPL allows configuration via properties (key-value pairs), like for the serialization method that is used, reliability, message ordering, performance monitoring support, etc. Whereas the layers on top of IPL request certain properties, the Ibis instantiation is using local configuration files containing information about the locally available functionality (like being reliable or unreliable, or having ordered broadcast communication). At startup, Ibis tries to load each of the implementations listed in the file, and checks if they adhere to the required properties, until all requirements have been met. If this is impossible, properties are re-negotiated with the layers on top of IPL.

3.2.1 Non-functional properties of Ibis

Performance
For Ibis, performance is the paramount design criterion. The IPL provides communication primitives using send ports and receive ports. A careful design of these ports and primitives allows flexible communication channels, streaming of data, efficient hardware multicast and zero-copy transfers.

The layer above the IPL creates send and receive ports, which are connected to form a *unidirectional message channel*, see Figure 5. New (empty) message objects can be requested from send ports, and data items of any type can

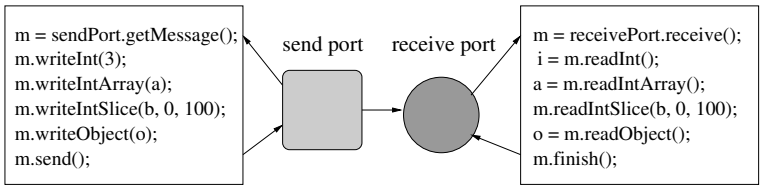

Figure 5: Ibis send ports and receive ports

be inserted incrementally, allowing for streaming of long messages. Both primitive types and arbitrary objects can be written. When all data is inserted, the *send* primitive can be invoked on the message.

The IPL offers two ways to receive messages, the port's blocking *receive* primitive, as well as upcalls, providing the mechanism for implicit message receipt.

Fault tolerance
Currently, the Ibis runtime system provides transparent fault tolerance for the Satin divide-and-conquer API (see below) [45]. Fault tolerance for other programming models is subject to ongoing work.

Security and trust
Ibis is focusing on communication between processes. Currently, it supports encrypted communication by incorporating the secure socket layer (SSL) in its communication protocol stack [17]. Issues like authentication and authorization are beyond the scope of Ibis.

Platform independence
Platform independence is mostly addressed by using Java. Besides, the IPL's dynamic loading facility hides many properties of the underlying resources and networks from the application.

3.2.2 Functional properties of Ibis

Access to compute resources, job spawning and scheduling
The API's provided by Ibis focus on communication between parallel processes. As such, they do not provide access to compute resources, job spawning, or scheduling systems. Instead, Ibis assumes that underlying Grid middleware takes care of scheduling and starting an Ibis application. The Ibis model does not prevent settings with processes joining and leaving running applications, however, such dynamically changing process groups are not exposed to the application.

Access to file and data resources
As Ibis is hiding the Grid environment from the application, its API's do not provide access to Grid file and data resources either. As with compute resources, Ibis assumes underlying middleware (like the Java GAT [4]) to provide such functionality, if needed.

Communication between parallel and distributed processes
As shown in Figure 4, Ibis provides a set of communication API's on top of the IPL. Besides the API's shown in the figure and discussed below, Ibis also supports CCJ [31], a simple set of collective communication operations, inspired by MPI [30]. An implementation of the MPJ [15] message passing API has been added recently. Support for GridSuperscalar [5] is subject to ongoing work.

RMI For basic, object-based communication, Ibis provides Java's standard *remote method invocation* (RMI) [42]. The Ibis RMI implementation is optimized for high performance between remote processes [42].

RepMI Using RMI for globally shared objects can lead to serious performance degradation as almost all method invocations have to be performed remotely across a Grid network. For applications with high read/write ratios to their shared objects, replicated objects can perform better, as all read operations become local, and only write operations need to be broadcast to all replicas, as implemented in *replicated method invocation* (RepMI) [28].

With RepMI, groups of replicated objects are identified by the programmer using special marker interfaces (like with Java RMI). The Ibis runtime system works together with a byte code rewriter (similar to RMI's *rmic*) that understands the special marker interfaces and generates communication code for the respective classes.

GMI Compared to RepMI, *group method invocation* (GMI) [29] can provide more relaxed consistency among *object groups*, favoring higher performance and much better flexibility. GMI is a conceptual extension of RMI, also allowing groups of *stubs* and *skeletons* to communicate, allowing RMI-like *group* communication. In GMI, methods can be invoked either on a single stub, or collectively on a group of stubs. Likewise, invocations can be handled by an individual skeleton or a group of skeletons. Complex communication patterns among stubs and skeletons can be deployed, depending on the communication semantics. Schemes for method invocation and result handling can be combined orthogonally, providing a wide spectrum of operations that span, among others, both synchronous and asynchronous RMI, future objects, and collective communication as known from MPI.

Satin Divide-and-conquer parallelism is provided using the Satin interface [41], shown in Figure 6. The application first extends the *Spawnable* marker interface, indicating to the byte code rewriter to generate code for parallel execution. An application class then extends the *SatinObject* class and implements its marker interface, together tagging the recursive invocations as asynchronous. The *sync* method blocks until all spawned invocations have returned their result.

```
interface FibInterface extends ibis.satin.Spawnable {
          public long fib(long n);
}

class Fib extends ibis.satin.SatinObject implements FibInterface {
      public long fib(long n) {
             if(n < 2) return n;
             long x = fib(n-1); /* spawn, tagged in FibInterface */
             long y = fib(n-2); /* spawn, tagged in FibInterface */
             sync();            /* from ibis.satin.SatinObject   */
             return x + y;
      }
}
```

Figure 6: Satin code example for the Fibonacci numbers

Each spawned method invocation creates an invocation record that is stored locally in a work queue. Idle processors obtain work by an algorithm called *cluster-aware random work stealing* (CRS). It has been shown in [43] that CRS can execute parallel applications very efficiently in Grid environments while the application code is completely shielded from Grid peculiarities by the Satin interface.

Application monitoring and steering Application monitoring and steering are not provided explicitly by Ibis; this is subject to ongoing work.

3.3 GAT

The Grid Application Toolkit (GAT) aims to enable scientific applications in grid environments. It helps to integrate grid capabilities in application programs, by providing a simple and *stable* API with well known API paradigms (e.g. POSIX like file access), interfacing to grid resources and services, abstracting details of underlying grid middleware. This allows to interface to different versions or implementations of grid middleware without any code change in the application.

To illustrate how the GAT can be used, Figure 7 shows a complete C++ program performing a remote file access – it reads the file given as command line argument and prints its content on *stdout*.

```
#include <iostream>
#include <GAT.hpp>
int main (int argc, char** argv) {
  try {
    GAT::Context    context;
    GAT::FileStream file (context, GAT::Location (argv[1]));
    char buffer[1024] = "";

    while ( 0 < file.Read (buffer, sizeof (buffer) - 1) )
      std::cout << buffer;
  }
  catch (GAT::Exception const &e) {
    std::cerr << e.GetMessage() << std::endl;
  }
  return (0);
}
```

Figure 7: GAT example in C++ for a `cat`.

To the application, the file access method actually used is completely transparent – it could be ssl, ftp, grid-ftp or libc. The GAT takes care of translating these calls to the appropriate middleware operations, by preserving the defined semantics.

3.3.1 Functional properties of GAT

The GAT API specification covers the six following areas:

(1) physical files
(2) logical files
(3) communication
(4) compute resources
(5) monitoring and steering
(6) persistent meta data and information

These areas have been derived from application use cases in the GridLab [3] project, and are not intended to provide complete coverage of grid capabilities. Hence, GAT is not intended to cover *every* application use case in grids (although the GAT authors tried to extend the scope to other probable use cases).

Simplicity is *the* major constraint for the API specification, along with the requirement to reuse well known API paradigms wherever possible. These constraints lead to the API as described in more detail in [4].

The GAT API addresses all functional properties as listed in Section 2.2. However, there is one notable exception: GAT does not provide any high-performance inter process communication. GAT's communication mechanisms are aimed at application steering and control and support only simple pipes. High-performance communication between parallel processes is considered to be beyond the scope of GAT.

Architecture The GAT architecture [4] follows 2 major design goals: (a) the API layer is to be *independent* from the grid environment; and (b) bindings to the grid environments must be exchangeable/extendable at runtime, on user and/or administrator level. This implies that the API layer cannot implement the API capabilities directly, but has to dynamically dispatch the calls to a lower, exchangeable layer. That principle is reflected in the API as shown in Figure 8.

A thin API layer with GAT syntax interfaces to the application. The calls are forwarded to the GAT engine, which dynamically dispatches the API calls to the adaptor layer. Adaptors are modules which implement the semantics of the call and bind the GAT to specific a grid middleware.

Recently, the Global Grid Forum (GGF) has formed a research group (SAGA-RG – Simple API for Grid Applications) to standardize a grid API. The GAT group is actively taking part in this group. In fact, the SAGA design will be very similar to the GAT design. However, SAGA prescribes only the API, not the architecture of any implementation.

3.3.2 Non-functional properties of GAT

Performance In grid environments, network latency and remote service delays form a major performance problem. Compared to that, any overhead imposed by the local implementation is small. Hence, the GAT engine's dynamic system of adaptor selection and call dispatching introduces little overhead, when compared to the total call execution

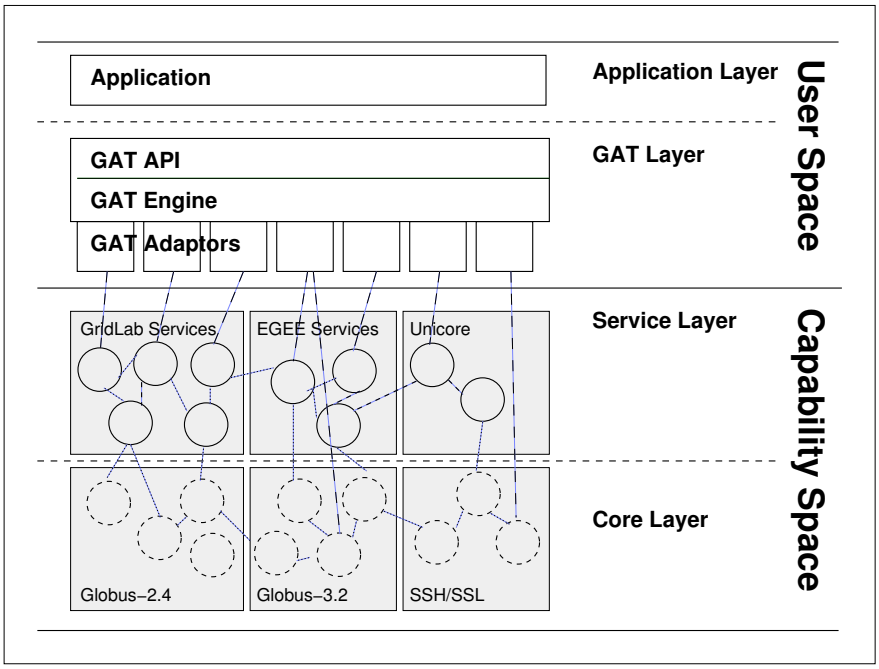

Figure 8: GAT Architecture: the engine dispatches API calls to adaptors, which bind to a specific grid capability provider.

time. However, GAT can be configured to make optimal use of available resources. For example, a system administrator may want to install/configure a file movement adaptor using a locally-available high-speed LAN. Also, as in all grid applications, adaptor-level caching mechanisms are encouraged to minimize network latencies.

Fault Tolerance Upon failure of a grid operation, the GAT engine is falls back to other adaptors. For example, if a GSI adaptor fails to create a coomunication channel, a SSH adaptor would be tried, and may succeed. That process is transparent to the application.

By abstracting the grid operations from the application, fault tolerance is much easier to provide, and can be hidden from the end user. But error reporting and auditing are crucial for the user. GAT supports both, and returns a hierarchical stack trace of call information for all operations performed.

On adaptor level, fault tolerance is implemented in whichever manner is suitable for the grid environment the adaptor binds to.

Security and Trust GAT can only be as secure as its adaptors – the engine itself does not perform remote operations and cannot directly enforce any security. However, the engine provides the means to implement a coherent security model across all adaptors. Also, a minimal security level (`local, ssh, gsi` etc.) can be specified via user and system preferences, and prevents less secure adaptors from being used. Ultimately, the trust relationship lies with the adaptors, and with the grid middleware, rather than with GAT. GAT merely delegates, capabilities as well as security.

Platform Independence Platform, environment and language independence has been a major objective of the GAT implementation. The GAT reference implementation is written in ANSI-C/ISO-C99, is natively developed on Windows and Linux, and has been ported to MacOS-X, Solaris, IRIX, Linux-64 and other UNIX's. A native Java GAT implementation is also available, which by design is platform independent.

The GAT architecture makes it usable on any grid environment providing a set of adaptors. GAT comes with a set of local adaptors, binding the API to the *libc* – hence it is possible to develop complex grid applications on disconnected systems, and to later run the same executable on a full scale grid.

GAT adaptors exist to GridLab services [3], to Globus (pre Web Services) [39], to ssh/ssl/sftp, and to the Unicore Resource Broker [18]. Ongoing work is addressing adaptor development for specific experiments or projects, like to the Storage Resource Broker (SRB) [8], or for dynamic light-path allocation for file transfer. The Java GAT implementation uses the CoG Toolkit [44] to bind to various Globus incarnations.

3.4 Summary

We summarize the comparison of our three systems in Table 1. There, bullet points indicate properties that are addressed while hollow circles refer to unaddressed properties. From the table it becomes obvious that the three systems have been designed for somewhat different purposes. These choices directly influence which (functional and non-functional) properties are addressed, actually.

Property	ProActive	Ibis	GAT
Non-Functional Properties			
performance	●	●	○
fault tolerance	●	●	●
security / trust	● / ○	● / ○	● / ○
platform independence	●	●	●
Functional Properties			
resources / job spawning / scheduling	● / ● / ○	○ / ○ / ○	● / ● / ●
files / data resources	○ / ○	○ / ○	● / ●
parallel / distributed communication	● / ●	● / ●	○ / ●
application monitoring / steering	● / ○	○ / ○	● / ●

Table 1: Comparison of the three frameworks

4 Related work

Besides the ones presented, there are various other grid programming environments. First of all, one has to name the "native" grid APIs and environments, such as Condor [20] and Globus [39]. These systems reach deep into the fabric layer of the grid, but also provide programming interfaces for higher levels. However, these interfaces are often conceived as being unfit for application development, in terms of complexity and dependency on the actual middleware and its configuration.

The Java CoG [44] has been an early valiant effort to offer low level grid capabilities to applications. First, CoG has been a wrapper around early Globus versions. The CoG became a reimplementation of large parts of Globus, to get independent of the Globus development cycle. Nowadays, CoG has a very similar architecture to GAT, and to the general architecture presented in Section 5. CoG is, as GAT, one of the major supporters for the GGF SAGA API effort.

Early versions of the Java CoG have been binding directly and exclusively to a specific Globus version. In fact, that approach was taken in other projects as well, with the main objective to abstract the complexity of grid programming for the end user. However, that approach can not be kept in sync with neither the dynamic grid environment, nor with the progress of the grid middleware development.

Various MPI implementations aim at supporting large parallel applications in grids. MPICH-G [19] is using Globus communication facilities. PACX-MPI [26] is an MPI implementation which can efficiently handle WAN connectivity. Both packages (and others with the same scope) are widely used in the community.

Remote steering has been an interesting target for distributed applications for a long time. It seems to be one of the areas which could benefit significantly from the use of grid paradigms – the absence of a suitable distributed security framework has hindered the widespread use of remote steering until now. The Reality Grid Project [13] is providing one example of a grid-based steering infrastructure, based on the concepts of OGSA and Web Services. The Reality Grid Project is also one of the supporters of the GGF SAGA API.

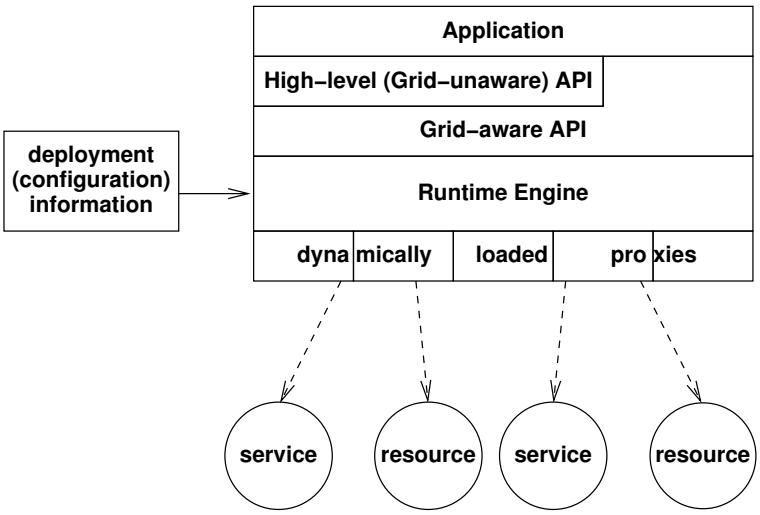

Figure 9: Generic runtime architecture model

ICENI [21] is a grid middleware framework with an added value to the lower-level grid services. It is a system of structured information that allows to match applications with heterogenous resources and services, in order to maximise utilisation of the grid fabric. Applications are encapsulated in a component-based manner, which clearly separates the provided abstraction and its possibly multiple implementations. Implementations are selected at runtime, so as to take advantage of dynamic information, and are selected in the context of the application, rather than a single component. This yields to an execution plan specifying the implementation selection and the resources upon which they are to be deployed. Overall, the burden of code modification for specific grid services is shifted from the application designer to the middleware itself.

5 Generic architecture model

Our three systems, ProActive, Ibis, and GAT, provide API functionality that partially overlaps, and partially complements each other. A much stronger similarity, however, can be observed from their software architectures, which is due to the non-functional properties of platform independence, performance, and fault tolerance. These properties strongly call for systems that are able to dynamically adjust themselves to the actual grid environment underneath.

Figure 9 shows the generic architecture for grid application programming environments that can address the properties identified in Section 2.

- Application code is programmed exclusively using the API's provided by the envisioned grid application programming environment. We distinguish between grid-aware (lower level) and grid-unaware (higher level) API's. Both kinds of API's will be needed, depending on the purpose of the application. For example, one kind of application might simply wish to use some invisible computational "power" provided by the grid (as intended by the power grid metaphor) while others might want to interact explicitly with specific resources like given databases or scientific instruments.

- The API's are implemented by a *runtime engine*. The engine's most important task is to delegate API invocations to the right service or resource. In the presence of transient errors and of varying performance, this delegation becomes a non-trivial task. In fact, the runtime engine should implement sophisticated strategies to bind requests to the best-suited service. Whereas possibly many services might fulfill a given request, the choice among them is guided purely by non-functional properties like service availability, performance, and security level.

- Delegation to a selected service can be achieved by dynamically loaded proxies. Dynamic binding is important for separating application programs from the actual grid execution environments. This way, applications can also be executed in new (versions of) environments that came to existence only after an application has been

written. Besides for platform independence, this dynamic binding is necessary for handling of transient error conditions at runtime.

- For resource and service selection purposes, the runtime engine needs configuration information to provide the right bindings. Current implementations (like ProActive, Ibis, and GAT) mostly rely on configuration files, provided by system administrators, describing the properties of given machines in a grid. However, some configuration information is of more dynamic nature and requires dynamic monitoring and information services to provide up-to-date information. Examples are access control for users and resources, and performance data about network connections that might impact resource selection.

It is obvious that current grid application programming environments comply only partially to this architecture. However, with the advent of more sophisticated grid middleware, like grid component architectures, or widely deployed monitoring and information services, also programming environments will be able to benefit and provide more flexible, better performing, and failure-resilient services to applications.

6 Conclusions and future directions

Grids can be considered as distributed systems for which heterogeneity, wide-area distribution, security and trust requirements, failure probability, as well as latency and bandwidth of communication networks are exacerbated. Such platforms currently challenge application programmers and users. Tackling these challenges calls for significantly advanced application programming environments.

We have identified a set of functional and non-functional properties of such future application programming environments. Based on this set, we have analyzed existing environments, emphasizing ProActive, Ibis, and GAT, which have been developed by the authors and their colleagues, which we also consider to be among the currently most advanced systems.

Our analysis has shown that none of our systems curently addresses all properties. This is mostly due to the different application scenarios for which our systems have been developed. Based on our analysis, we have identified a generic architecture for future grid programming environments that allows building systems that will be capable of addressing the complete set of properties, and will thus be able to overcome today's problems and challenges.

A promising road to implementing our envisioned grid application programming environment is to explore a component-oriented approach, as also proposed in [40]. To date, GridKit [16] seems to be unique in effectively applying such a component-based approach to both the runtime environment and the application layer of a grid platform. Most existing component-based systems address applications only, like [1, 21] or the implementation of Fractal components using ProActive [10].

The inherent openness, introspection and reconfiguration capabilities offered by a component-oriented approach appear promising for implementing both grid programming environments and applications that are portable, adaptive, self-managing, and self-healing. Implementing such properties will require advanced decision taking and planning inside the runtime engine. Applying a component-oriented approach will thus complement the generic architecture identified in this chapter. Components will be the building blocks that ae assembled and reassembled at run time, yielding flexible grid application environments.

acknowledgments

This manuscript would not have been possible without the many contributions of our past and present colleagues. We would like to thank all the major contributors in the design and development of ProActive: by contribution order, J. Vayssière, L. Mestre, R. Quilici, L. Baduel, A. Contes, M. Morel, C. Delbé, A. di Costanzo, V. Legrand, G. Chazarain. We owe a lot to the Ibis team: Jason Maassen, Rob van Nieuwpoort, Ceriel Jacobs, Rutger Hofman, Kees van Reeuwijk, Gosia Wrzesinska, Niels Drost, Olivier Aumage, and Alexandre Denis. We also express our thanks to the GAT designers and writers. GAT was designed by Tom Goodale, Gabrielle Allen, Ed Seidel, John Shalf and others. The C Version has mainly been implemented by Hartmut Kaiser, who also wrote the C++ and Python wrappers. The Java version was written by Rob van Nieuwpoort. We would like to thank our many colleagues from the EU GridLab project.

The current collaboration is carried out in part under the FP6 Network of Excellence CoreGRID funded by the European Commission (contract IST-2002-004265). ProActive is supported by INRIA, CNRS, French Ministry of

Education, DGA, through PhD funding, and ObjectWeb, ITEA OSMOSE, France Telecom R&D. Ibis is supported by the Netherlands Organization for Scientific Research (NWO) and the Dutch Ministry of Education, Culture and Science (OC&W), and is part of the ICT innovation program of the Dutch Ministry of Economic Affairs (EZ). GAT has been supported via the European Commission's funding for the GridLab project (contract IST-2001-32113).

References

[1] M. Aldinucci, M. Coppola, S. Campa, M. Danelutto, M. Vanneschi, and C. Zoccolo. Structured iplementation of component based grid programming environments. in this volume.

[2] W. Allcock, J. Bester, J. Bresnahan, A. Chervenak, L. Liming, S. Meder, and S. Tuecke. GridFTP Protocol Specification. GGF GridFTP Working Group Document, 2002.

[3] Gabrielle Allen, Kelly Davis, Konstantinos N. Dolkas, Nikolaos D. Doulamis, Tom Goodale, Thilo Kielmann, Andre Merzky, Jarek Nabrzyski, Juliusz Pukacki, Thomas Radke, Michael Russell, Ed Seidel, John Shalf, and Ian Taylor. Enabling Applications on the Grid - A GridLab Overview. *International Journal on High Performance Computing Applications*, 17(4):449–466, 2003.

[4] Gabrielle Allen, Kelly Davis, Tom Goodale, Andrei Hutanu, Hartmut Kaiser, Thilo Kielmann, Andre Merzky, Rob van Nieuwpoort, Alexander Reinefeld, Florian Schintke, Thorsten Schütt, Ed Seidel, and Brygg Ullmer. The Grid Application Toolkit: Towards Generic and Easy Application Programming Interfaces for the Grid. *Proceedings of the IEEE*, 93(3):534–550, 2005.

[5] Rosa M. Badia, Jesús Labarta, Raül Sirvent, Josep M. Pérez, José M. Cela, and Rogeli Grima. Programming Grid Applications with GRID Superscalar. *Journal of Grid Computing*, 1(2):151–170, 2003.

[6] L. Baduel, F. Baude, and D. Caromel. Object-Oriented SPMD. In *CCGrid 2005*, 2005.

[7] H.E. Bal, J.G. Steiner, and A.S. Tanenbaum. Programming Languages for Distributed Computing Systems. *ACM Computing Surveys*, 21(3):261–322, 1989.

[8] Chaitanya Baru, Reagan Moore, Arcot Rajasekar, and Michael Wan. The sdsc storage resource broker. In *CASCON '98: Proceedings of the 1998 conference of the Centre for Advanced Studies on Collaborative research*, page 5. IBM Press, 1998.

[9] F. Baude, D. Caromel, F. Huet, L. Mestre, and J. Vayssière. Interactive and Descriptor-Based Deployment of Object-Oriented Grid Applications. In *HPDC-11*, pages 93–102. IEEE Computer Society, July 2002.

[10] F. Baude, D. Caromel, and M. Morel. From distributed objects to hierarchical grid components. In *DOA*, volume 2888, pages 1226–1242. LNCS, 2003.

[11] M. Beck, J. Dongarra, J. Huang, T. Moore, and J. Plank. Active Logistical State Management in the GridSolve/L. In *Proc. 4th International Symposium on Cluster Computing and the Grid (CCGrid 2004)*, 2004.

[12] R. Berlich, M. Kunze, and K. Schwarz. Grid Computing in Europe: From Research to Deployment. *CRPIT series, Proceedings of the Australasian Workshop on Grid Computing and e-Research (AusGrid 2005)*, 44, Jan. 2005.

[13] J. M. Brooke, P. V. Coveney, J. Harting, S. Jha, S. M. Pickles, R. L. Pinning, and A. R. Porter. Computational Steering in RealityGrid. In *Proceedings of the UK e-Science All Hands Meeting 2003*, 2003.

[14] D. Caromel. Towards a Method of Object-Oriented Concurrent Programming. *Communications of the ACM*, 36(9):90–102, September 1993.

[15] Bryan Carpenter, Vladimir Getov, Glenn Judd, Anthony Skjellum, and Geoffrey Fox. MPJ: MPI-like message passing for Java. *Concurency: Practice and Experience*, 12(11):1019–1038, 2000.

[16] G. Coulson, P. Grace, P. Blair, and al. Component-based Middleware Framework for Configurable and Reconfigurable Grid Computing. *To appear in Concurrency and Computation: Practice and Experience*, 2005.

[17] Alexandre Denis, Olivier Aumage, Rutger Hofman, Kees Verstoep, Thilo Kielmann, and Henri E. Bal. Wide-Area Communication for Grids: An Integrated Solution to Connectivity, Performance and Security Problems. In *Proc.HPDC-13*, pages 97–106, 2004.

[18] Dietmar Erwin, editor. *Joint Project Report for the BMBF Project UNICORE Plus*. UNICORE Forum e.V., 2003.

[19] Ian Foster and Nicholas T. Karonis. A grid-enabled mpi: message passing in heterogeneous distributed computing systems. In *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, pages 1–11. IEEE Computer Society, 1998.

[20] James Frey, Todd Tannenbaum, Miron Livny, Ian Foster, and Steven Tuecke. Condor-g: A computation management agent for multi-institutional grids. *Cluster Computing*, 5(3):237–246, 2002.

[21] N. Furmento, A. Mayer, S. McGough, S. Newhouse, T . Field, and J. Darlington. ICENI: Optimisation of Component Applications within a Grid Environment. *Parallel Computing*, 28(12), 2002.

[22] The Global Grid Forum (GGF). http://www.gridforum.org/.

[23] Wolfgang Hoschek, Javier Jaen-Martinez, Asad Samar, Heinz Stockinger, and Kurt Stockinger. Data Management in an International Data Grid Project. In *Proc. IEEE/ACM International Workshop on Grid Computing (Grid'2000)*, 2000.

[24] Fabrice Huet, Denis Caromel, and Henri E. Bal. A High Performance Java Middleware with a Real Application. In *SuperComputing 2004*, 2004.

[25] N. Karonis, B. Toonen, and I. Foster. MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface. *Journal of Parallel and Distributed Computing*, 2003.

[26] Rainer Keller, Edgar Gabriel, Bettina Krammer, Matthias S. Müller, and Michael M. Resch. Towards efficient execution of MPI applications on the Grid: porting and optimization issues. *Journal of Grid Computing*, 1(2):133–149, 2003.

[27] Thilo Kielmann, Rutger F.H. Hofman, Henri E. Bal, Aske Plaat, and Raoul A.F. Bhoedjang. MagPIe: MPI's Collective Communication Operations for Clustered Wide Area Systems. In *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'99)*, pages 131–140, 1999.

[28] Jason Maassen, Thilo Kielmann, and Henri E. Bal. Parallel Application Experience with Replicated Method Invocation. *Concurrency and Computation: Practice and Experience*, 13(8–9):681–712, 2001.

[29] Jason Maassen, Thilo Kielmann, and Henri E. Bal. GMI: Flexible and Efficient Group Method Invocation for Parallel Programming. In *Proc. LCR '02: Sixth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, Washington, DC, 2002. To be published in LNCS.

[30] Message Passing Interface Forum. MPI: A Message Passing Interface Standard. *International Journal of Supercomputing Applications*, 8(3/4), 1994.

[31] Arnold Nelisse, Jason Maassen, Thilo Kielmann, and Henri E. Bal. CCJ: Object-based Message Passing and Collective Communication in Java. *Concurrency and Computation: Practice and Experience*, 15(3–5):341–369, 2003.

[32] Jason Novotny, Michael Russell, and Oliver Wehrens. GridSphere: A Portal Framework for Building Collaborations. In *1st International Workshop on Middleware for Grid Computing*, Rio de Janeiro, 2003.

[33] Jennifer M. Schopf, Jarek Nabrzyski, and Jan Weglarz, editors. *Grid resource management: state of the art and future trends*. Kluwer, 2004.

[34] H. Sivakumar, S. Bailey, and R. L. Grossman. PSockets: The Case for Application-level Network Striping for Data Intensive Applications using High Speed Wide Area Networks. In *Proc. Supercomputing (SC2000)*, 2000.

[35] O. Smirnova, P. Eerola, T. Ekelof, M. Elbert, J.R. Hansen, A. Konstantinov, B. Konya, J.L. Nielsen, F. Ould-Saada, and A. Waananen. The NorduGrid Architecture and Middleware for Scientific Applications. In *ICCS 2003*, number 2657 in LNCS. Springer-Verlag, 2003.

[36] Y. Tanaka, H. Nakada, S. Sekiguchi, T. Suzumura, and S. Matsuoka. Ninf-G: A Reference Implementation of RPC-based Programming Middleware for Grid Computing. *Journal of Grid Computing*, 1(1):41–51, 2003.

[37] Ian Taylor, Matthew Shields, Ian Wang, and Omer Rana. Triana Applications within Grid Computing and Peer to Peer Environments. *Journal of Grid Computing*, 1(2):199–217, 2003.

[38] The GEO600 project. http://www.geo600.uni-hannover.de/.

[39] The Globus Alliance. http://www.globus.org/.

[40] J. Thiyagalingam, S. Isaiadis, and V. Getov. Towards Building a Generic Grid Services Platform: a component-oriented approach. In V. Getov and T. Kielmann, editors, *Component Models and Systems for Grid Applications*. Springer Verlag, 2005.

[41] Rob V. van Nieuwpoort, Thilo Kielmann, and Henri E. Bal. Efficient Load Balancing for Wide-area Divide-and-Conquer Applications. In *Proc. PPoPP '01: ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 34–43, 2001.

[42] Rob V. van Nieuwpoort, Jason Maassen, Gosia Wrzesinska, Rutger Hofman, Ceriel Jacobs, Thilo Kielmann, and Henri E. Bal. Ibis: a Flexible and Efficient Java-based Grid Programming Environment. *Concurrency and Computation: Practice and Experience*, 17(7–8):1079–1107, 2005.

[43] Rob V. van Nieuwpoort, Jason Maassen, Gosia Wrzesinska, Thilo Kielmann, and Henri E. Bal. Adaptive Load Balancing for Divide-and-Conquer Grid Applications. *Journal of Supercomputing*, accepted for publication, 2004.

[44] Gregor von Laszewski, Ian Foster, Jarek Gawor, and Peter Lane. A Java Commodity Grid Kit. *Concurrency and Computation: Practice and Experience*, 13(8-9):643–662, 2001.

[45] Gosia Wrzesinska, Rob V. van Nieuwpoort, Jason Maassen, and Henri E. Bal. Fault-tolerance, Malleability and Migration for Divide-and-Conquer Applications on the Grid. In *19th International Parallel and Distributed Processing Symposium (IPDPS 2005)*, Denver, USA, 2005.