# NETIBIS: An Efficient and Dynamic Communication System for Heterogeneous Grids

Olivier Aumage*        Rutger Hofman        Henri Bal

Vrije Universiteit, Amsterdam, The Netherlands
http://www.cs.vu.nl/ibis/

## Abstract

*Grids are more heterogeneous and dynamic than traditional parallel or distributed systems, both in terms of processors and of interconnects. A grid communication system must handle many issues: first, it must run on networks that are not yet determined when the application is launched, including user-space interconnects; second, it must transparently run on different networks at the same time; third, it should yield performance close to that of specialized communication systems.*

*In this paper, we present* NETIBIS, *a new Java communication system that provides a uniform interface for any underlying intercluster or intracluster network.* NETIBIS *solves the heterogeneity issues posed by Grid computing by dynamically constructing network protocol stacks out of* drivers, *self-contained building blocks for flexible configuration, with limited functionality per driver.*

*We describe the design and implementation of the major* NETIBIS *drivers for serialization, multicast, reliability, and various underlying networks: TCP, UDP and GM (user-space Myrinet network layer). We also describe various optimizations for performance, like layer collapsing for the GM driver.*

*From experiments with benchmarks and applications on various networks on a Linux cluster, we find that* NETIBIS *performance is quite close to that of efficient alternatives, and its RMI implementation far outperforms traditional Sun RMI.*

*Further experiments on a European Grid show that* NETIBIS *fully meets the demands of Grid computing. It transparently carries the software engineering burden of multiple, heterogeneous networks, while yielding performance that is very close to the best available.*

## 1. Introduction

Grid computing poses new challenges on network research, as grids are more heterogeneous and dynamic than traditional parallel or distributed systems. For example, a distributed supercomputing application should ideally be able to run on any grid resource (e.g., a cluster, supercomputer, or shared-memory machine). The local interconnect that the application is going to use typically is not known at the time the application is launched, let alone at the time it is written. Moreover, such applications should be able to use multiple different resources at the same time. An easy way out of this problem is to always use the TCP protocol.

With high-speed interconnects, however, low-level protocols (such as GM on Myrinet) are much more efficient.

Efficient and flexible networking support for grids thus is inherently more difficult than for parallel or distributed systems. A communication system for the Grid therefore should have the following properties:

- it must be dynamic, so it can run on networks that are not yet determined when the application is launched; the protocol stack therefore should be *runtime configurable*;
- it must be heterogeneous, so it can run on multiple different networks at the same time;
- it must be efficient, so it can optimally exploit any fast local networks of the grid resources on which the application will run.

Most existing communication systems have only some of these properties (exceptions are discussed in Section 6).

In this paper, we present a new communication system called NETIBIS that combines all three properties. NETIBIS is part of the Ibis system, which is a Java-centric programming environment for grid computing [20]. The key idea in Ibis is to write virtually all systems software in Java, making it easy to run in a heterogeneous environment. To obtain high performance, Ibis allows native (non-Java) code as special-case optimizations. Also, Ibis performs several important optimizations using bytecode transformations. For example, Ibis uses bytecode rewriting to optimize away the overhead of serialization, which hampers many other Java implementations [16].

Ibis and NETIBIS have been used for many realistic applications [11] and have been used for Grid experiments at a fairly large (European) scale. Also, Ibis has been used to study interconnection problems (e.g., firewalls) in such environments [7]. In addition, several programming systems (e.g., ProActive and Satin) have been implemented with Ibis. This paper focuses on the design, implementation, and performance of NETIBIS.

The structure of this paper is as follows. Section 2 provides background information on Ibis. Section 3 describes the design of NETIBIS and Section 4 discusses several implementation aspects. In Section 5, we evaluate the performance of NETIBIS using benchmarks and applications that run on a single cluster and on a grid. Section 6 discusses related work and Section 7 presents our conclusions.

---

*Currently at Inria LaBRI, Talence, France
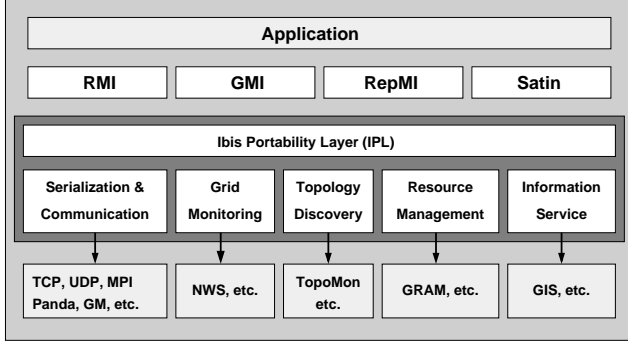
**Figure 1. Design of Ibis. The various modules can be loaded dynamically, using runtime class loading.**

## 2. Ibis

Ibis [20] is a Java-centric environment for grid programming. Ibis uses Java's "write-once, run-everywhere" property to address the intrinsic heterogeneity of grids. Virtually all Ibis communication software and runtime systems are implemented in Java, and Ibis runs out-of-the-box on heterogeneous grids, such as the European GridLab testbed [20]. A major research problem studied in the Ibis project is how to implement this Java-centric approach efficiently.

The structure of the Ibis system is shown in Figure 1. The Ibis Portability Layer (IPL) is a thin interface. The IPL can have different implementations, that can be selected and loaded into the application *at run time*. The IPL defines serialization and communication and provides interfaces to grid services such as topology discovery and monitoring. Ibis currently implements three application programming models on top of IPL: remote method invocation (RMI), group method invocation (GMI), and divide-and-conquer parallelism (Satin). All of them have efficient implementations for grids. An implementation of RepMI (replicated objects) is under way; also ProActive [11] has been implemented using Ibis.

Unlike many message-passing systems, the IPL has no concept of hosts or threads, but uses location-independent *Ibis identifiers* to identify Ibis nodes. A registry, called *Ibis Name Service*, is provided to locate receive ports, allowing to bootstrap connections.

### 2.1. Ibis communication

The IPL provides one basic communication abstraction, *unidirectional message channels*. Endpoints of communication are *send ports* and *receive ports*. As shown in previous work, constructing message channels from send and receive ports is highly flexible, and allows streaming of data and zero-copy transfers [20]. Figure 2 shows such a channel together with IPL code to send and to receive a message.

To support group communication, a send port can be connected to multiple receive ports: *multicast*, and vice
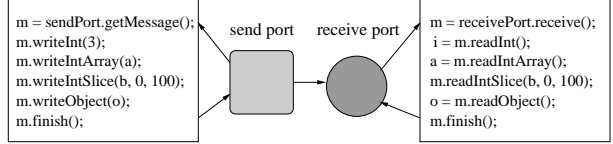


**Figure 2. The IPL communication abstraction.**

versa: *multireceive*. Even any $n$-to-$m$ connection between send and receive ports is possible. For further structuring, IPL ports are typed using properties (key-value pairs). Only ports of the same type may be connected to each other.

### 2.2. Ibis serialization

Object serialization is known to be a major bottleneck in parallel Java applications. Ibis addresses this performance problem by providing its own serialization, which is much more efficient than traditional serialization mechanisms. The latter mechanisms use reflection at runtime; Ibis serialization moves this work to compile time as much as possible. Ibis rewrites bytecode of serializable classes to add class-specific (de)serialization methods. This optimization is similar to that of our Manta [16] system, but now using Java bytecode instead of native code, making the new implementation much more portable.

Traditional serialization generates a byte stream. In contrast, Ibis serialization generates buffers of primitive types, which can be transported more efficiently by some networks. However, if the network interface protocol supports only transmission of byte arrays, the primitive-type buffers are efficiently converted to byte buffers using Java *NIO* (New I/O package). NIO was introduced in Java 1.4 and offers different *views* of buffers of primitives. For example, a buffer can be viewed as a byte array or as a double array, and this allows conversion of primitive types to bytes at the speed of memory copy.

In detail, Ibis serialization works as follows. Objects are serialized by first writing a one-word object identifier to handle duplicate objects in the message, then a one-word class identifier (which is negotiated the first time an object of this class is sent over the current connection), and then the object fields. Primitive-type fields are written to a buffer of the corresponding type; object-type fields are handled recursively. The buffers that contain primitive-type values and arrays are either streamed unchanged to the network or mass-converted to bytes.

## 3. Design of NETIBIS

NETIBIS is an implementation of the Ibis IPL that combines runtime configurability and network heterogeneity management with efficient communication. NETIBIS is able to establish communication dynamically over hardware links ranging from high-speed local interconnects (e.g., Myrinet and shared memory) to wide-area networks. The
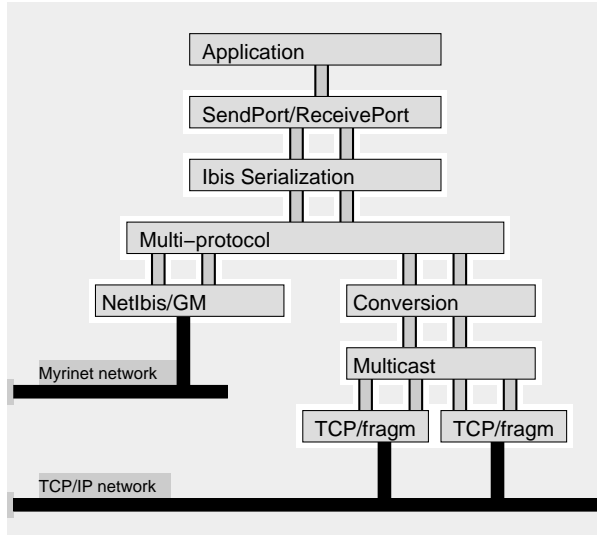
**Figure 3. The** NETIBIS **architecture.**

communication connections can be configured dynamically at connection establishment time.

The key idea to achieve both efficiency and runtime configurability is to use protocol modules that can be stacked onto each other, and possibly collapsed if performance requires it. Each protocol module provides its own functionality such as *serialization* or *multicast* communication, or access to some networking interface or hardware (e.g., TCP, UDP, or Myricom's messaging layer GM for Myrinet).

NETIBIS supports protocol stacking using two basic concepts: connections and drivers. The *connection* links a send port to a receive port. Connections are end-to-end from the IPL send port down through the NETIBIS modular protocol stack to the actual hardware link on the sending machine and up through the NETIBIS stack again on the receive side to the IPL receive port.

A protocol module in the communication stack is called a *driver*. A driver consists of an *Output* object at the sender side or an *Input* object at the receiver side. For each driver type, there is a global *Driver* object that manages all global resources of this driver type, and acts as a factory to create Input and Output objects when a connection is established. Figure 3 illustrates a NETIBIS protocol stack. It shows that a driver may possess multiple incoming or outgoing connections. Each send port and receive port may have its own stack configuration, but a connection may only be established between ports of the same stack configuration.

Configuration of port types can be specified by the user in a number of ways: as an attribute of a new port type; in a configuration file; or, for simple port configuration, on the command line.

### 3.1. Connections

A connection in NETIBIS is a unidirectional virtual FIFO networking link from a source to a destination process.

NETIBIS connections actually are made of two network links. The *application* link is a unidirectional link and uses the networking software/hardware selected by the user. It is exclusively used by the application. The *service* link is a bi-directional link made of a pair of streams (TCP streams in the current NETIBIS implementation). It may be used by the NETIBIS internals to exchange data between the source and destination nodes of the connection and for synchronization. It is also used for dynamic negotiation, e.g. for buffer sizes, and it provides a basic means of connection failure detection. A third category of network links, called *bootstrap links* [7], may be used when establishing a connection is difficult, for instance when firewalls or NAT (Network Address Translation) are involved.

### 3.2. Drivers

We distinguish between two groups of drivers, the *filter* drivers and the *network* drivers. Filter drivers implement some "high"-level protocol functionality while network drivers implement support for specific low-level network APIs and hardware. Filter drivers are internal nodes in the protocol stack, whereas network drivers are leaf nodes.

The classes that make up a driver are its Driver class, its Input class and its Output class. Per driver type, there is one global Driver instance that manages global resources for this driver. The Input and Output instances are responsible for managing the network connections. Each driver may control one or more connections. Each connection may go through one or more drivers before reaching the network.

The threefold organization of the NETIBIS drivers allows to flexibly distribute the implementation between Inputs/Outputs and the Driver. The Driver controls shared resource-based functionalities such as connection multiplexing and caching of resources (e.g., buffers) among connections, or implements network drivers such as GM which are built on one shared native implementation. On the other hand, features that do not require or profit from any global resources are implemented in the Input/Output in a straightforward fashion.

The interfaces of Driver, Input and Output classes are each identical for every driver. As a consequence, all drivers are used the same way and are therefore transparently exchangeable and stackable in arbitrary order. Network drivers are an exception because they always are at the lowest position in the stack. Not all conceivable protocol stack configurations make sense semantically. For instance, it is unnecessary to include more than one serialization driver, or a multicast driver if the IPL port is configured to use only unicast.

## 4. Implementation

Several drivers have been implemented so far and we describe the most important ones below. We distinguish between filter drivers and network drivers.

## 4.1. Filter drivers

The filter drivers each provide an optional *added value*. Driver functionality can be added or left out on demand in a NETIBIS protocol stack. Such flexibility allows grid applications to get most of the benefit of hand-customized communication stacks in a generic, portable and heterogeneity-safe way. Two filter drivers are not described here, because they are still under development: the security enforcement driver and the encryption driver.

**Serialization** The serialization drivers convert Java objects to byte buffers or buffers of primitive types. There are currently three serialization drivers: the *Sun* driver that uses Java's traditional serialization through Java's Object-Streams; the *Ibis* driver which implements Ibis serialization (see Section 2.2); and the *Data* driver that can transport (arrays of) primitive types but not general objects.

**Multi** We group a number of drivers here because they share much of their implementation; for software engineering reasons, the shared part is captured in a superclass, the generic *multi-driver* class. Their shared functionality is the capacity to fork the stack into multiple branches.

The *multicast/multireceive* driver implements the multicast and multireceive capacity of send and receive ports; outgoing messages are forwarded to each connection established through this stack, incoming messages are multiplexed by the Input that listens to each of its sub-Inputs. Each of the sub-Inputs and sub-Outputs must be of the same driver class.

The *multi-protocol* driver is capable of supporting different driver types for its sub-Inputs and sub-Outputs. It also forwards its outgoing messages to each sub-Output, and handles the possibly different message layout that may be required by the different sub-drivers.

The multi-driver lends itself to substantial optimizations with respect to a straightforward implementation. A multireceive Input for an explicit receive stack must spawn a thread for each sub-Input, and incur a thread switch on the critical path for each receive. If the port type is configured to use no multireceive, the multireceive driver is replaced by a no-op driver, and decoupling by means of a thread is unnecessary. The decoupling is also left out while there is only one connection *at runtime* through the multireceive driver. Equally, the multi-protocol Input needs decoupling only if there actually exist multiple types of connection from this one port, which happens only infrequently.

Another optimization is the possibility to integrate this driver into some network drivers (like the GM driver, see below) that can support the multicast/multireceive functionality internally at virtually zero cost.

**Reliability** This driver implements protocol reliability. It can be used, for example, with the UDP network driver, though it does not depend on any underlying protocol. It uses a sliding-window protocol; the window-size strategy can be easily replaced. The global Driver object of this driver type, which has knowledge of all connections through this layer, attempts to match an outgoing connection to each incoming connection between the same pair of hosts for carrying piggy-backed acknowledgements. This optimization can save much explicit acknowledgement traffic.

**Conversion** This driver implements conversion of (arrays of) primitive Java types into (arrays of) bytes with Java NIO.

**Fragmentation** The fragmentation driver implements packet fragmentation and reassembly if the lower drivers have a maximum packet buffer size.

**Multiplex** The multiplexing driver folds several connections into one. Demultiplexing must generally be done with a listening thread for each connection, which introduces a thread switch on the critical path. Whereas the multi driver splits the protocol stack, the multiplex driver merges it.

## 4.2. Network drivers

**TCP and UCP** The drivers for TCP and UDP are written entirely in Java, using Java Sockets. TCP socket links are bi-directional, whereas Ibis connections are unidirectional. Normally, TCP uses its back link to do piggy-backed acknowledgements, and thus can often save on explicit acknowledgements. To achieve the same effect, NETIBIS/TCP uses free TCP back links when a connection is set up between hosts that already have a TCP connection in the reverse direction.

**Shared memory** For multiprocessors, we have a pure-Java driver that implements the IPL message-passing primitives using shared memory.

**GM** GM is the protocol that is bundled with the Myrinet gigabit network. The GM driver uses a mixture of Java and C, because a Java implementation of the GM API does not exist (yet). For this driver, we implemented many optimizations. A multicast/multireceive and a multiplexing driver are integrated into NETIBIS/GM, at virtually zero cost in software engineering since thread handling must be implemented anyway in NETIBIS/GM. In our implementation, only one receiving thread polls the network through GM's single poll entry point, and other receiving threads wait until the polling thread wakes them up; in the frequent case that a polling thread is itself the intended receiver of the new message, no thread switch is incurred. Another optimization follows from the observation that in C, buffers of primitive types can be transmitted without conversion to bytes; this amounts to integrating a conversion driver into NETIBIS/GM. For large messages, it switches to a rendez-vous protocol that allows zero-copy transfers.

**Future drivers** Currently, we are implementing a TCP driver using NIO. The main advantages are the possibility to transfer primitive types without conversion to bytes and the presence of a `select` call, which offers the possibility to integrate a multiplexing driver. Other useful extensions of our work would be drivers for other networks (e.g., InfiniBand) and a driver on top of MPI.

| Paradigm | network | lat. | throughput (MBit/s) | | | |
|---|---|---|---|---|---|---|
| | | $\mu$s | byte | int | double | tree |
| Socket | 100Mb | 128 | 90 | – | – | – |
| TCPIBIS | 100Mb | 133 | 85 | 83 | 83 | 62 |
| NETIBIS | 100Mb | 139 | 84 | 82 | 81 | 50 |
| Socket | IP/Myri | 96 | 750 | – | – | – |
| TCPIBIS | IP/Myri | 100 | 610 | 350 | 340 | 159 |
| NETIBIS | IP/Myri | 106 | 540 | 470 | 450 | 115 |
| PANDAIBIS | GM/Myri | 44 | 940 | 920 | 920 | 230 |
| NETIBIS | GM/Myri | 42 | 1100 | 1060 | 1060 | 230 |

**Table 1. RPC performance**

| Paradigm | network | lat. | throughput (MBit/s) | | | |
|---|---|---|---|---|---|---|
| | | $\mu$s | byte | int | double | tree |
| Sun | 100Mb | 259* | 68 | 66 | 66 | 16 |
| TCPIBIS | 100Mb | 142 | 83 | 82 | 81 | 64 |
| NETIBIS | 100Mb | 156 | 78 | 78 | 78 | 50 |
| Sun | IP/Myri | 223* | 300 | 140 | 112 | 26 |
| TCPIBIS | IP/Myri | 102 | 370 | 250 | 260 | 150 |
| NETIBIS | IP/Myri | 117 | 250 | 250 | 230 | 111 |
| PANDAIBIS | GM/Myri | 45 | 430 | 440 | 420 | 210 |
| NETIBIS | GM/Myri | 46 | 420 | 420 | 420 | 150 |

**Table 2. RMI performance**

# 5. Performance evaluation

The primary design consideration for NETIBIS was dynamic configurability and heterogeneity management. To be useful for real applications, however, NETIBIS performance must be at least reasonable in comparison with other networking paradigms; this is true both for the situation that the application will run on hardware that is unknown in advance, and for the situation that the application will run on multiple clusters with different networks, within or between clusters. We will investigate NETIBIS performance at three levels. First, microbenchmarks will show the null-latency and large-message throughput, with comparison measurements of other, relevant paradigms. Second, we show performance of a number of applications, some of them application kernels (SOR), others real-world applications. Third, we investigate performance of one application in wide-area experiments and describe an application that uses NETIBIS to do remote visualisation.

All experiments were run on the DAS system, which consists of one cluster of 72 nodes at the Vrije Universiteit, Amsterdam and three clusters spread over the Netherlands, each of 32 nodes. Each DAS cluster consists of 1GHz Pentium IIIs, running RedHat Linux 7.2 and connected by 100Mbit Ethernet and 2Gbit Myrinet. The experiments on the DAS were done with the IBM 1.4.1 JVM, the fastest JVM available for Linux. The last experiment on a distributed system was run on several GridLab clusters.

## 5.1. Microbenchmarks

One comparison platform for IPL benchmarks is socket performance in Java, both over 100Mbit Ethernet and over Myricom's kernel implementation of IP over Myrinet. The speed of sockets in Java is very close to sockets in C, for which we don't present numbers in the table. Benchmark performance is also compared with already existing implementations of the Ibis interface: TCPIBIS for TCP runs, and PANDAIBIS for GM over Myrinet. These latter implementations were both well-tuned for their respective networks. TCPIBIS uses the TCP protocol everywhere, so it runs on heterogeneous grids, but it cannot benefit from high speed interconnects with dedicated low-level protocols. PANDAIBIS is implemented on the Panda [2] communication layer and is not dynamically reconfigurable, because it is mostly written in native code.

Table 1 shows that TCPIBIS adds less than $5\mu$s to the basic Java socket latency. NETIBIS adds about $10\mu$s; this difference shows that the overhead costs of layer traversal in NETIBIS are small. The optimization to use free TCP back links for reverse Ibis connections so TCP can piggyback its acknowledgements (both in TCPIBIS and NETIBIS/TCP) turns out to save $15$-$20\mu$s on an RPC. PANDAIBIS and NETIBIS/GM are comparable in latency. This is a reflection of the extensive layer collapsing in NETIBIS/GM.

The throughput measurements use arrays of size 100,000 bytes. Byte arrays are transmitted without serialization. The other throughput measurements are all done with Ibis serialization. Except for the tree datatype, all come close to saturation of the 100Mbit network. Java NIO is used to convert ints and doubles to bytes for TCP; before the introduction of NIO in Java 1.4, serialization of especially doubles was expensive. *Tree* (a binary tree of 1023 nodes, with in the nodes 4 integers, a left and a right pointer) incurs considerable overhead; the TCP implementations do not support true streaming: TCP flow control allows little pipelining in the three stages (serialization, transmission, deserialization).

On IP/Myrinet, the superior throughput for arrays of ints and doubles is due to a greater packet size in NETIBIS, 32KB versus 2KB for TCPIBIS that was tuned for 100Mbit Ethernet. Byte arrays are sent over the network without any buffering, so there throughput is somewhat lower for NETIBIS than for TCPIBIS. The same holds for trees. As with latency, the cause is layer traversal overhead.

The Ibis implementations over GM support streaming. NETIBIS/GM achieves higher throughput than PANDAIBIS due to its rendez-vous mechanism, which PANDAIBIS lacks.

Besides IPL benchmarks, we present the performance of NETIBIS RMI benchmarks in Table 2, in comparison with traditional Sun RMI, which runs both on 100Mbit Ethernet and on IP over Myrinet. Another RMI implementation, both over IP and over GM, is KaRMI [17], which has a reputed performance improvement over traditional Sun RMI. A KaRMI null-RMI over GM took $44\mu$s, comparable to the Ibis GM implementations. The Sun latency numbers that are marked with * are measured in a different way; the average latency of 10000 RMIs was 3–4ms, due to occasional roundtrips that took 250ms (these occurred for all JVMs that we tested, IBM 1.4, Sun 1.4, Sun 1.5). We attribute this to buggy interaction with RedHat 7.2 kernel threads; the problem did not occur on RedHat Enterprise kernels, which use
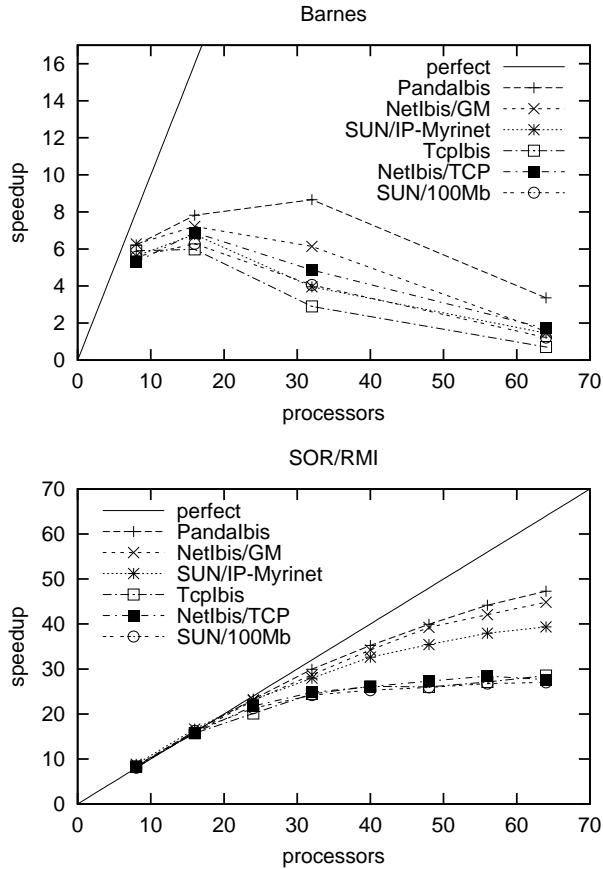
**Figure 4. LAN speedup for RMI applications**



**Figure 5. LAN speedup for Satin applications**

a different threads implementation. We present the latency with the exceptional roundtrip numbers filtered out.

Like the IBIS implementations, traditional Sun uses NIO to convert to bytes. This is reflected in the throughput of ints and doubles. The difference between Sun and IBIS for the tree throughput clearly shows the advantages gained by Ibis serialization.

## 5.2. LAN applications

Parallel performance of a networking system is not determined only by latency and maximum throughput. Other factors can also be important, dependent on application properties: send and receive overhead; throughput for small messages; and, for user-space networks, the strategy of integration between network polling and the thread system. We evaluate the performance of NETIBIS for a number of applications. Sequential execution time is given in Table 3; speedup figures are presented in Figures 4, 5, and 6.

**Barnes** is an RMI implementation of Barnes-Hut, following the highly efficient parallelization by Blackstone and

| Application | Paradigm | data set | Time (s) |
|---|---|---|---|
| Barnes | RMI | 50,000 | 82 |
| SOR | RMI | 8192x8192 | 4539 |
| Satisfiability | Satin | fpga10_11_uns_rcr | 2215 |
| Raytracer | Satin | balls2_medium | 3200 |
| SOR | IPL | 4096x4096 | 668 |

**Table 3. Applications: Sequential execution time**
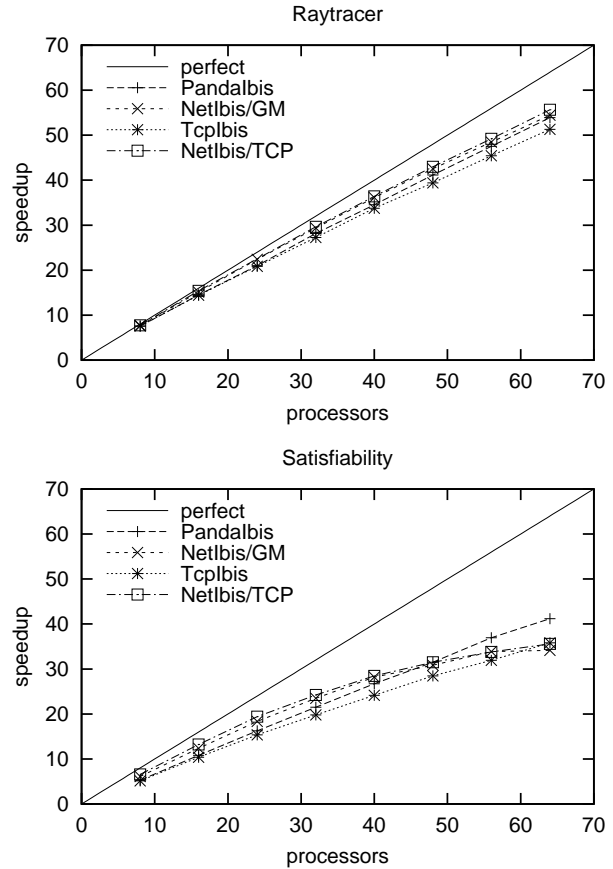
Suel [4]. It turned out that RMI is really unfit for achieving good performance for this algorithm because it lacks asynchronous communication and is unable to send slices of arrays. It is expected that an implementation on top of Ibis IPL, which does offer these communication primitives, can be efficient. Nevertheless, we chose to include this application with a relatively small data set, because it does show performance differences; as we would expect from the microbenchmarks, the Ibis versions on GM/Myrinet perform best. Unexpectedly, TCPIBIS is even slower than traditional Sun over 100MBit.

**SOR/RMI** performs red/black Successive Overrelaxation. In each iteration, neighbors exchange one row, and a reduce-to-all is performed to detect termination. Parallel performance is dominated by the reduce-to-all. The time taken by the reduce-to-all increases with the number of machines, whereas the computation time decreases. The result is that speedup flattens off, for Ethernet beyond 32 nodes at this data size, for Myrinet beyond 64 nodes. If the reduce is removed from the calculation, this application scales much better (not shown in the graphs).

**Raytracer** is an application that renders a scene with many balls in Satin, a Divide and Conquer system on top of Ibis [21]. Speedup is calculated with respect to a sequential version with all Satin code compiled out. Although there is a small sequential stage in the code, speedup is close to
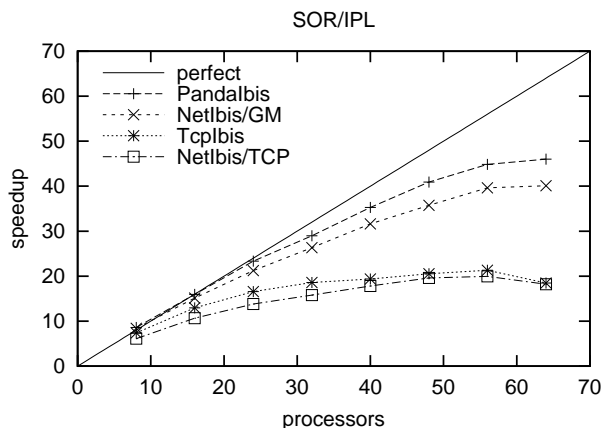
**Figure 6. LAN speedup for an IPL application**

perfect. There is little difference in performance between the networks; the Satin paradigm is very insensitive to latency, and the bandwidth requirements of this application are modest.

**Satisfiability** solves the satisfiability problem in Satin to verify a CPU architecture [22]. This application has more runtime overhead than Raytracer, because it copies its state in the search tree before every potential job spawn. The speedup in Figure 5 is again with respect to a sequential version with Satin support compiled out. However, if speedup is calculated with respect to a sequential version that includes Satin calls and state copying, parallel efficiency becomes close to perfect. Like Raytracer, performance differences for the different networks are small.

**SOR/IPL** is another implementation of red/black SOR, this time on top of the Ibis IPL. It uses asynchronous messages and a spanning-tree reduce. Although it uses a smaller data set than SOR/RMI, speedup for the GM/Myrinet Ibis implementations is comparable. The efficiency still highly depends on latency.

In all applications, the performance of NETIBIS/GM is slightly below that of PANDAIBIS, although the performance figures of the microbenchmarks are almost the same. The performance differences for applications is caused by two factors: the send/receive overhead of NETIBIS/GM is somewhat higher and the polling strategy in PANDAIBIS is somewhat more advanced.

### 5.3. Applications on a WAN

Of the many possible applications of a Grid, we chose to present two examples. The first uses a cluster computer for carrying out a calculation and a workstation to do on-line, remote visualization. The second is a parallel application that runs in a distributed fashion on a number of clusters, connected by wide-area links.

We also suggest that, besides the types of application investigated below, another class of applications will profit from the intelligent auto-configuration of NETIBIS: workflow applications, that are in essence pipelines of heterogeneous computation stages. An example is the Triana pro-

gramming paradigm. Some stages in a workflow application might be communication-intensive, like FFTs; between stages of the pipeline, low latency is usually not of crucial importance. Combining fast intra-cluster networking with portable inter-cluster networking was also achievable before, e.g. by loading a PANDAIBIS for inside the cluster and a TCPIBIS for between clusters. The programmer had to be aware at all network invocations which Ibis must be selected, and message forwarding between different Ibises must be programmed by hand. The advantage of NETIBIS over this approach is that the programmer (of application or RTS) can remain oblivious of network heterogeneity, and just invokes the Ibis IPL.

**Visualization** The SOR/RMI application was run on 32 nodes of the DAS cluster at the Vrije Universiteit. For the visualization workstation, we chose a PC running Windows XP at one of the authors' home. The PC was connected to the internet through a NAT switch and a 1024/512 ADSL link. The visualization program sampled the data state of each of the worker processes in a tight loop; the worker nodes each used an RMI object to create and export a down-sampled view of the data. The NETIBIS configuration for both the SOR/RMI program and the visualization program was identical; it included a multi-protocol driver with GM for data exchange between the worker nodes and TCP for the visualization. The low speed of the ADSL link severely limited display refresh rate.

Although this is a restricted example, it illustrates well how the Grid-induced flexibility of NETIBIS can be put to use for remote tasks like visualization or monitoring, while the computation itself can calculate at the full speed of a cluster and its high-speed local interconnect. The advantage of NETIBIS here is even greater than in workflow applications. The application programmer has no control over which network is used, and traditional RMI necessarily uses just TCP.

**Application on the DAS multi-cluster** In our experience, several applications can be optimized to run efficiently on wide-area systems (and Grids), for example by doing latency hiding or message combining on the wide-area messages. Such optimized applications sometimes are insensitive to wide-area latency and bandwidth, but still communicate intensively over the local interconnect. We used the SOR implementation on top of Ibis IPL, which supports asynchronous message passing. To improve wide-area performance, we optimized the reduce-to-all operation that dominates parallel performance so that only one wide-area latency is involved [14].

Table 4 shows that speedup for the multi-protocol version is clearly better than for the TCP versions. The numbers presented are the minimum time over tens of runs for each data point. The variation of runtime is much larger than we are used to within the cluster, so that some individual run timings for a multi-protocol run are considerably

| Cluster configuration | TCPIBIS | NETIBIS | | PANDAIBIS |
|---|---|---|---|---|
| | | TCP | multi-protocol | |
| 1x64 | 36.2 | 36.7 | 16.7 | 14.5 |
| 2x32 | 30.9 | 35.7 | 22.3 | n/a |
| 4x16 | 27.4 | 31.4 | 22.3 | n/a |

**Table 4. Execution time in seconds of SOR/IPL on the DAS multicluster.**

slower than some individual run timings of the TCP-only versions. Both average and minimum time are lower for the multi-protocol version. The TCP multi-cluster runs are faster than the single-cluster runs. This is attributed to the modified reduce implementation; the single-cluster implementation uses a spanning tree, which performance-wise strikes a balance between latency and throughput. The modified version reduces latency and also reduces throughput, which, for messages of only one double, yields better performance.

**Application on the Grid** We also ran SOR/IPL for a number of IBIS implementations on collections of clusters within GridLab at various clusters over Europe (Linux/TCP clusters in Hungary, at Sara in Amsterdam, and the wide-area DAS). Here, the variation of processor speed and local and interlocal interconnect was so large that we did not obtain a performance improvement from using Myrinet at some of the clusters. A general conclusion might be that homogeneous applications that profit from fast local networks only in part of the multi-cluster are probably rare. If the application at hand is in that class, NETIBIS provides the desired combination of flexibility and performance.

## 6. Related work

The idea of using *protocol composition* for setting up customized communication stacks out of basic building blocks is definitely not new. What distinguishes NETIBIS from other projects is that it is completely in user space, that it is Java-specific and (Grid) application oriented, and focuses equally on efficiency and runtime configurability.

As early as 1984, the Streams framework introduced modular, dynamically configurable stacks into the Unix kernel [19], with a focus on character devices.

The SILK project integrated Scout paths, i.e. modular protocol stacks, inside the Linux kernel [3].

The Horus/Ensemble projects [10, 23] from Cornell University also promoted the use of stacked micro-protocol modules. Their focus is on group communication protocols, which are more heavy-weight than our NETIBIS protocols.

The x-Kernel project at the University of Arizona also used a modular protocol composition approach [12, 18]. This project was more specifically targeted towards low-level network protocol implementations such as TCP/IP stacks. It was designed to work inside operating systems and originally was a standalone operating system.

Both Horus/Ensemble and the x-Kernel emphasize the use of protocol composition for implementing security. Both were mentioned in a preliminary Grid study [8], whose authors express concerns about the overhead of the protocol composition technique; however, they suggest that the high flexibility of such a modular model for building communication stacks would be likely to compensate it.

Recent research on software routers, active networks and packet filtering also proposes modular, dynamic configuration of network stacks [5, 6, 15]. These stacks may run on network hardware, in the kernel and partly in user space.

Many projects have also studied how to extend traditional distributed computing network interfaces like MPI to grid-enabled state by integrating mechanisms to handle grid heterogeneity (e.g., [1, 9, 13]). Of the requirements outlined for NETIBIS in the introduction, all these implementations address heterogeneity, some also address efficiency between heterogeneous platforms, but the requirement of runtime configurability is not addressed by any of these implementations or the MPI standard.

## 7. Conclusion

Grids are more heterogeneous than parallel and distributed systems in terms of processors, internal networks and wide-area interconnects. Therefore, a grid communication system must address more issues. First, it must run on networks that are not yet determined when the application is launched. Second, it must transparently support running on different networks at the same time. Third, its efficiency should match specialized local communication systems.

In this paper, we presented NETIBIS, a new Java communication system that provides a uniform interface for any underlying communication system, whether local or heterogeneous and distributed. NETIBIS implements the IBIS interface, which was designed for Java-centric Grid computing. Our solution to the grid network requirements is to use *drivers*, self-contained building blocks for flexible configuration, and dynamically build a fitting protocol stack to provide the desired functionality.

From experiments on one Linux cluster, we found that NETIBIS provides performance that is quite close to that of specialized, already existing Ibis implementations on slow and fast networks, inside and out of the kernel (TCP, GM over Myrinet). The RMI implementation in NETIBIS is considerably faster than traditional Sun RMI. From microbenchmarks and application measurements, we conclude that NETIBIS comes quite close to giving the best available performance on single-cluster runs where the relative speed of processor and network is unknown beforehand, by dynamically selecting the appropriate driver stack. The software overhead introduced by the highly structured driver engineering is limited, and in some cases was easily optimized away by driver collapsing.

NETIBIS also yields high performance for applications that run on multiple, possibly heterogeneous clusters by

configuring multi-protocol drivers. It allows to transparently use a fast local interconnect, possibly in user space, as well as standard wide-area networks with TCP.

It also allows to run applications like work flows, which may require intensive communication in some stages in combination with low communication requirements between different stages, on (possibly) heterogeneous hardware without any penalty in software engineering, and with the smallest possible penalty in performance.

In conclusion, NETIBIS truly delivers on the promise of Java Grid Computing.

## Acknowledgements

## References

[1] O. Aumage and G. Mercier. MPICH/MadIII: a Cluster of Clusters Enabled MPI Implementation. In *CCGrid 2003*, pages 26–35, Tokyo, May 2003. IEEE.

[2] H.E. Bal, R. Bhoedjang, R. Hofman, C. Jacobs, K. Langendoen, T. Rühl, and M.F. Kaashoek. Performance evaluation of the Orca shared-object system. *ACM Transactions on Computer Systems*, 16(1):1–40, February 1998.

[3] A. Bavier, Th. Voigt, M. Wawrzoniak, L. Peterson, and P. Gunningberg. SILK: Scout Paths in the Linux Kernel. Technical Report 2002-009, Department of Information Technology, Uppsala University, Sweden, February 2002.

[4] D. Blackston and T. Suel. Highly portable and efficient implementations of parallel adaptive n-body methods. In *SuperComputing 1997*, pages 1–20. ACM Press, 1997.

[5] H. Bos, W. de Bruijn, M. Cristea, T. Nguyen, and G. Portokalidis. FFPF: Fairly Fast Packet Filters. In *Proceedings of OSDI'04*, San Francisco, CA, December 2004.

[6] H. Bos and B. Samwel. The OKE Corral: Code Organisation and Reconfiguration at Runtime using Active Linking. In *Proceedings of IWAN'2002*, Zuerich, Switzerland, December 2002.

[7] A. Denis, O. Aumage, R. Hofman, K. Verstoep, Th. Kielmann, and H.E. Bal. Wide-area communication for grids: An integrated solution to connectivity, performance and security problems. In *13th HPDC*, Honolulu, Hawaii, USA, Jun 2004.

[8] I. Foster, N. Karonis, C. Kesselman, and C. Tuecke. Managing security in high-performance distributed computing. *Cluster Computing*, 1(1):95–107, 1998.

[9] E. Gabriel, M. Resch, T. Beisel, and R. Keller. Distributed Computing in a Heterogeneous Computing Environment. In V. Alexandrov and J. Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, LNCS, pages 180–188. Springer, 1998.

[10] M. Hayden. The Ensemble System. Technical Report TR98-1662, Cornell University, January 1998.

[11] F. Huet, D. Caromel, and H.E. Bal. A high performance Java middleware with a real application. In *SuperComputing 2004*, Pittsburgh, Pennsylvania, USA, 2004. ACM/IEEE.

[12] N. C. Hutchinson and L. L. Peterson. The x-Kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

[13] N. Karonis, B. Toonen, and I. Foster. MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface. In *Journal of Parallel and Distributed Computing*, volume 63, pages 551–563, May 2003.

[14] Th. Kielmann, R. Hofman, H.E. Bal, A. Plaat, and R. Bhoedjang. MAGPIE: MPI's Collective Communication Operations for Clustered Wide Area Systems. In *PPoPP 1999*, pages 131–140, Atlanta, GA, May 1999.

[15] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M.F. Kaashoek. The Click modular router . *ACM Transactions on Computer Systems*, 8(3):263–297, August 2000.

[16] J. Maassen, R. van Nieuwpoort, R. Veldema, H.E. Bal, T. Kielmann, C. Jacobs, and R. Hofman. Efficient Java RMI for parallel programming. *ACM Trans. Program. Lang. Syst.*, 23(6):747–775, 2001.

[17] Chr. Nester, M. Philippsen, and B. Haumacher. A more efficient RMI for Java. In *Proceedings of the ACM 1999 Conference on Java Grande*, pages 152–159, San Fransisco, Jun 1999. ACM.

[18] S. W. O'Malley and L. L. Peterson. A dynamic network architecture. *ACM Transactions on Computer Systems*, 10(2):110–143, May 1992.

[19] Dennis M. Ritchie. A Stream Input-Output System. *AT&T Bell Labs Technical Journal*, 63(8):1897–1910, 1984.

[20] R. van Nieuwpoort, J. Maassen, R. Hofman, Th. Kielmann, and H.E. Bal. Ibis: a Flexible and Efficient Java-based Grid Programming Environment. *Concurrency & Computation: Practice & Experience*, 16:1–29, 2002.

[21] R. van Nieuwpoort, J. Maassen, G. Wrzesinska, Th. Kielmann, and H.E. Bal. Satin: Simple and Efficient Java-based Grid Programming. *Journal of Parallel and Distributed Computing Practices*, accepted for publication, 2004.

[22] C. van Reeuwijk, R. van Nieuwpoort, and H.E. Bal. Developing Java Grid Applications with Ibis. *Available online at http://www.cs.vu.nl/~reeuwijk/publications/ibis-grid-apps.pdf*, Nov 2004.

[23] R. van Renesse, K.P. Birman, R. Friedman, M. Hayden, and D.A. Karr. A framework for protocol composition in Horus. In *14th PODC*, pages 80–89, Ottawa, Ottawa, Canada, august 1995.