A Distributed Parallel Programming Framework

Nenad Stankovic and Kang Zhang, *Senior Member*, *IEEE*

Abstract—This paper presents Visper, a novel object-oriented framework that identifies and enhances common services and programming primitives, and implements a generic set of classes applicable to multiple programming models in a distributed environment. Groups of objects, which can be programmed in a uniform and transparent manner, and agent-based distributed system management, are also featured in Visper. A prototype system is designed and implemented in Java, with a number of visual utilities that facilitate program development and portability. As a use case, Visper integrates parallel programming in an MPI-like message-passing paradigm at a high level with services such as checkpointing and fault tolerance at a lower level. The paper reports a range of performance evaluation on the prototype and compares it to related works.

Index Terms—Distributed parallel programming, visual programming, message-passing, object-oriented model, fault tolerance.

1 Introduction

Existing distributed programming systems support a fixed number of computing models (often only one), making them hard or impossible for tailoring to specific needs of an application that was not originally considered. Building a completely new system is difficult and there is need for a generic framework with services common to different software models and applications. Rather than developing applications directly on top of the transport layer, the goal of this research is a layered and extensible framework that supports specific models through simple customization. When enhanced with visual programming techniques, such a middleware should facilitate writing of parallel and distributed applications. We have investigated the framework concepts by implementing a new, distributed, object-oriented (OO) environment called *Visper*.

OO programming is highly regarded when dealing with complex systems due to extensibility, maintainability, and reusability of components. Historically, the parallel programming community has been slow in adopting object orientation, due to high cost and time requirements when building new or replacing existing applications. Therefore, a new infrastructure was often forced to compromise between serving the legacy system and language and facilitating the new development. As a result, the realized solution was just a bridge between old practices and new features. However, as we have learned in our survey of related works, this does not necessarily yield a better or more natural solution. Pasadena Working Group 7 [27] has been discussing the OO paradigm and techniques and promoting them as a more productive approach to parallel software. Unfortunately, the Pasadena recommendations also advocate an evolutionary rather than a revolutionary approach.

- *N. Stankovic is with Nokia, 5 Wayside Road, Burlington, MA 01803. E-mail: Nenad.Stankovic@nokia.com.*
- *K. Zhang is with the Dept. of Computer Science, University of Texas at Dallas, Richardson, TX 75083-0688. E-mail: kzhang@utdallas.edu.*

Manuscript received 16 Oct. 2000; revised 26 Apr. 2001; accepted 16 May 2001.
Recommended for acceptance by Luqi.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 113000.

We believe that efficiency and scalability in process instantiation and communication, portability, fault tolerance, extensibility, and interoperability, as discussed at Pasadena, can be achieved without compromising the new development. Rather, a new system should be transparent and open to legacy or other code through a well-defined and standard interface. It should comply with the new (e.g., OO) methodology, rather than making new techniques comply with old practices. This sets out the general objectives of our work. We will consider asynchronous distributed systems. Specifically, distributed means that the processors are physically separated and the processes executing in the system communicate by passing messages along channels. Asynchronous means that the system has no global clock and that there are no bounds on relative local clock speeds, execution speeds, or message transmission delays.

The rest of the paper is organized as follows: The next section overviews the major design features in Visper. Section 3 describes the visual development method and a new visual formalism adopted. Section 4 outlines the structure and main components of the Visper prototype. The OO parallel programming support, fault-tolerant approach, and distributed management by software agents are detailed in Sections 5 to 7. Section 8 reports on the performance evaluation of the prototype, followed by a comparison to related works in Section 9. Section 10 concludes the paper.

2 Design Features

Distributed systems are complex software and hardware structures and their development is difficult and challenging. They should provide a consistent and uniform view of how to build and organize applications that run on them. A uniform model contributes to a single-system view. The issues the model has to address comprise scalability, heterogeneity, security, resource management, fault-tolerance, multilanguage support, extensibility, interoperability, and ease of use.

The idea behind Visper is in a seamless support for multiple programming models (e.g., parallel, sequential,

distributed, agents) that are, in their core, similar, if not identical, and can be realized with high-level tools. In fact, by definition, distributed and parallel programming differentiate against each other in terms of the hardware they use rather than applications they represent [8]. A set of cooperating mechanisms is needed that make up a reusable design through an OO approach and strict separation of concerns. Programs should use only those services that are essential to achieve the goal. Design can be customized for a specific application by choosing the appropriate implementations of objects and enhancing them with other objects, where necessary.

We followed this approach to enable flexible, transparent, and efficient interaction between application and system objects, regardless of their role or location. The framework is further characterized by its support for object (process) groups, distributed management by agents, fault tolerance, and visual program development.

2.1 Visual Program Development

Visualization is recognized as a powerful technique in understanding user requirements and helping software development since visual constructs and relationships are often easier to reason about than similar structures described in plain text. A visual language may enhance a programming language by providing intuitive graphical primitives and also enforce programming discipline with high-level compositional and structural constructs [5]. When developing a parallel program, four main stages can be identified: problem partitioning, program composition, debugging, and performance tuning [43]. Visual formalisms designed so far are often applicable to only some of the mentioned stages because the level of abstraction at one stage may not be appropriate at another.

We have designed a visual parallel programming language for graphical composition of message-passing programs. The graphical form in the language is scalable and hierarchical in specifying communication and synchronization. Due to the explicit visual representation of parallelism and nondeterminism, the potential interdependency and concurrency between processes are easily perceivable. The main advantage of our graphical form over the other existing forms is its consistent visual representation throughout all four stages that preserves the user's mental map [9].

2.2 Modular Programming and Object Groups

The goal of Visper is letting developers concentrate on designing and implementing functionality as objects in a modular manner, as this should be their main activity. Other issues, such as distribution and communication of objects, are different from what the object does (i.e., its *semantics*) and should be obtained from libraries. For example, a serious limitation of most current systems is the synchronous point-to-point (i.e., client-server) nature of their communication and coordination that scales up badly.

The collective state of a distributed application consists of the states of its semantics objects and, in our approach, replication and caching are important techniques to achieve scalability. Visper allows groups of distributed processes (as objects) to be collectively created and manipulated as single entities that cause minimal networking overhead and programming intervention. Further, the library of generic classes enables communication, coordination, and control of active program objects and object groups.

2.3 Fault Tolerance

A fault tolerance mechanism is vital in a network environment because it allows a failed application to be restarted from the most recent saved state, thus reducing rerun time by skipping over the already computed operations. Our fault tolerant model assumes that all processes in a distributed application are fail-stop [30], which means that program either completes and produces the correct result or execution terminates prematurely and the system notifies the user of this fact. The services to support fault tolerance include checkpointing, data recovery, detection of faulty hosts, and user notification. Visper implements a checkpointing mechanism that periodically saves the state of the objects to file on a per process basis. The Visper's layered checkpointing protocol blends naturally into the asynchronous execution model. It promotes fast program execution by implementing optimistic checkpointing, while avoiding an unbound rollback [29] through coordination. The proposed membership service delivers more accurate [14] and responsive identification of faulty versus slow processes.

2.4 Distributed Management by Agents

Agents are computational entities that act autonomously on behalf of other entities. Generally, agents perform their actions either proactively, reactively, or both, and may cooperate, learn, and become mobile [17]. As an open system, Visper is controlled and configured by a set of agents that provide the link between the system and application programs and enable separate development of system and programming services. They communicate by sending asynchronous messages that can be either active or passive. Active messages synthesize behavior or data or both to functionally enrich the system and the passive messages represent the basic vocabulary understood by all agents that belong to the same class. The active messages synthesize behavior or data, or both to functionally enrich the system.

3 Visual Parallel Programming

Our approach to visual parallel programming is based on a visual formalism called the *Process Communication Graph* (PCG) [32] that originates from the space-time diagram (STD) [24] and the concurrency map [37]. The latter two types of diagrams present the execution dynamics of parallel programs as a stream of events in a two-dimensional space where one axis represents the time and another the individual processes. They have been used for debugging and tuning of parallel programs, but not for composition. We have adopted a radical solution to extend the concept of these diagrams into composition phase, by redefining the time axis as a control flow axis and adding the concept of process groups to the process axis.

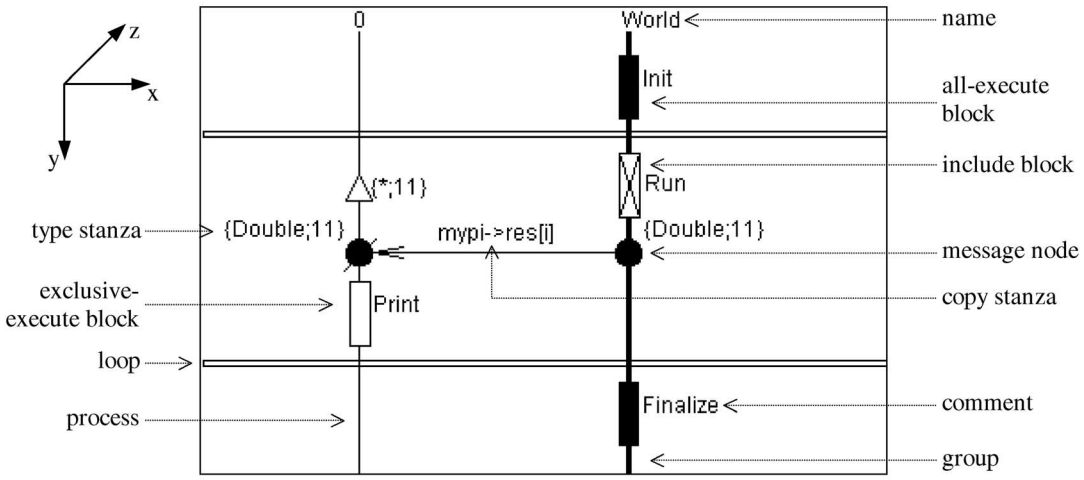


Fig. 1 Visual abstractions in PCG.

3.1 Process Communication Graph

Rather than viewing parallel programs as two-dimensional [2], the PCG provides a programmer with a three-dimensional programming space illustrated in an example in Fig. 1. The X-axis runs horizontally and is used to add processes and groups of processes to the programming space. Data are communicated among the processes along this axis. The Y-axis runs vertically from top to bottom and defines the execution sequence and time. The concept of groups of processes adds another dimension to parallel programming. The (virtual) Z-axis represents the number of processes in each group.

3.2 Visual Abstractions

Programming in PCG is not visual in all aspects, but exploits visual formalisms for data flow and parallel constructs. The visual abstractions in Fig. 1 can be classified in three-tiers that follow the way in which a parallel program is composed. The first tier comprises the processing unit called a *resource* that is uniquely identified by a *name*. The thin line represents an individual process, i.e., a group with only one process. The thick solid line represents a group of processes, manipulated as a single entity. Groups can be defined at compile time or dynamically allocated by a resource manager at runtime. These resources define the scope of the language primitives in the second tier.

The second tier comprises *basic symbols* for describing control and data flow in a message-passing program, all following the MPI standard [25]. Fig. 1 presents the main subset of the basic symbols defined in the PCG grammar. *Execute blocks* represent code or calls to routines written in C or Java. All resources in a program execute code found in an *all-execute block*. An *exclusive-execute block* is bound to a specific resource. A pair of *message nodes* describes a point-to-point message, a one-to-many or a many-to-one collective message. An *arc* designates the dataflow direction between two nodes. A *crossed* arc or message node represents a conditional message. A textual annotation attached to a communication primitive in the form of a stanza, further details the primitive, such as what type of data is communicated (*type stanza*) or what operation and data stores are involved (*copy stanza*).

The third tier consists of symbols that convey compositional information. An *include block* (the white rectangle with a cross) enables inclusion, by reference, of one graph into another. There is no limit imposed on the level of nesting. A pair of hollow horizontal lines represents an iterated computation (i.e., *loop*). A *comment* is any text that does not follow the syntax of the PCG grammar.

3.3 Program Composition

Parallel programs are large constructs with a structure that is sequentially ordered and spatially interdependent. Most often, the ratio of sequential to message-passing code favors the former, but the latter is harder to understand and optimize, and, thus, worth direct visual presentation. The PCG formalism is unique in recognizing conditional data flow rather than control flow, thus encouraging a flat data flow structure where process interdependencies are not deeply nested into the sequential control flow. Further, PCG does not define granularity at which execute blocks should be used, leaving that at a discretion of the programmer.

At the program composition level, PCG supports hierarchical program composition and reuse of modules by graph inclusion. Programmers can work concurrently on different modules and test them separately in Visper (Sections 5.1 and 8.1). At the design and programming level, the language empowers the programmer with the primitives that scale up [3], such as groups, execute blocks, and collective communications, when developing programs with a large number of resources. The tool has been used in teaching parallel programming, with students composing programs and visually analyzing the PCG and space-time diagram (STD) for communication patterns [23] and behavior [36].

Composing a PCG is performed visually in the PCG editor (Fig. 2). In an editing window (e.g., *CalculateX-Y.v* module of Monte Carlo program from Section 8.2), the programmer first draws processes and groups, thus defining the spatial program distribution and processes' roles, followed by other symbols to define processes'

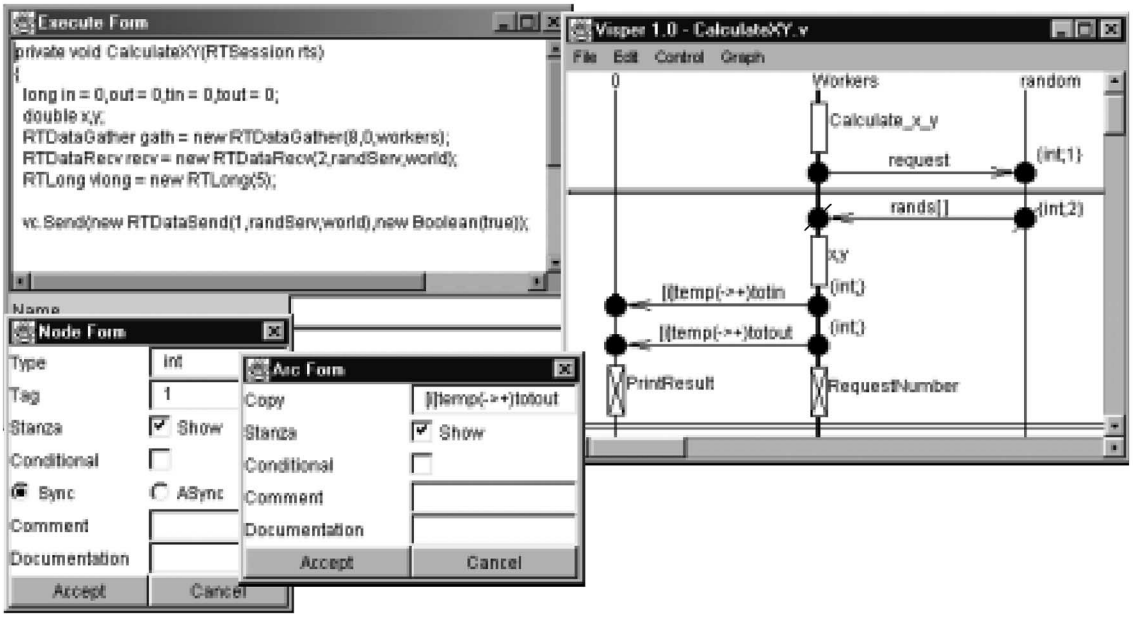


Fig. 2. PCG and its editor.

behavior (i.e., control and data flow aspects of an application). The module consists of a group called *Workers* and two processes, one of which collects and prints the result (i.e., *0*) and the other that generates random numbers called *random*. The whole scenario of receiving random numbers and performing the calculation is wrapped in a loop.

The programmer then annotates the graph by filling in a set of forms to describe the properties of the resources, execute blocks (an *ExecuteForm*), arcs, and nodes (e.g., *ArcForm* and *NodeForm*). For example, the programmer must specify such information as the sequential computation represented by the blocks and the types of the data sent in the messages. Finally, the programmer invokes a compile operation to automatically assemble a program. If no graphical syntax error has been detected, the program is composed and saved to a text file ready for compilation and execution.

4 The Prototype Architecture

The aforementioned design features of the Visper framework have been developed in a prototype [33] that is an interactive, object-oriented environment, with a set of tools for composition, execution, and testing of *single program multiple data* (SPMD) applications. Fig. 3 shows the major components of the prototype in two layers: *frontend* and *backend*. The frontend comprises the user interface that supports visual programming in PCG, configuration management, monitoring and debugging, and visualization of message-passing programs through its four tools. Independent from the frontend design, the backend implements the services that support the frontend and system services including the fault tolerant mechanism. This section gives an overview of the prototype with its structure and tools.

4.1 Frontend: User Interface Tools

The main functionality of the frontend is to support visual parallel program composition and visualization of program execution. It provides the means to compose, start, and analyze parallel programs through the *user interface*. Fig. 4 shows various frontend components. The top left window is the session dialog, the top right window is the debug window, and the bottom window is the console. By using the PCG editor (Fig. 2) in the *design tool,* the programmer draws a PCG to visually describe the structural aspects of an application, independent from the target architecture. The defined processes communicate by visually invoking message-passing primitives in the form of basic symbols. The behavioral aspects of processes are specified by program code. The design tool generates a structured internal representation of the computation, performs syntactic analysis of a constructed graph, and translates the graph into a parallel program.

The *configuration tool* informs the user about hosts that are available in a Visper-enabled *network of workstations* (NOW) and allows the user to configure the environment in terms of *sessions* in a session dialog. The user selects host names from the left list (e.g., *World*) and creates a new list on the right under a new session name. A session is an ordered set of hosts that represents a virtual parallel computer, i.e., a metacomputer. Once a session is defined and created, it is ready to accept requests for program execution. Each session can run only one program at a time. Sessions can grow or shrink since hosts can join or leave (e.g., crash). A session grows dynamically when a program allocates more resources via a resource manager (Section 5.2). The user can reconfigure an existing session by manually adding, removing, or reordering the hosts in a session dialogue. Through the *debugging tool*, the user enables and disables the recording of communication events. The recorded events, such as synchronization

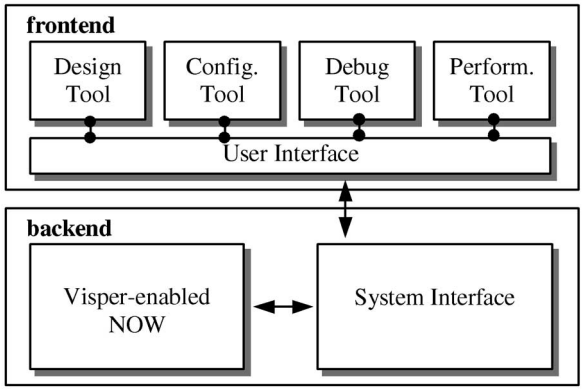


Fig. 3. Main components of the Visper prototype.

delays and communication ordering, can later be visualized in a space-time diagram and analyzed [23].

The top right subwindow in the console is a part of the *performance tool* that collects and displays the performance information (e.g., program execution time) and supports performance analysis with characteristic views (e.g., process activity). This subwindow also displays system-related events (e.g., user initiated reporting). The top left subwindow in the console displays the backend messages initiated by user actions (e.g., *J+* means that host *niaum00* has joined session *one* and its ID is *0*). The bottom subwindow displays program output or generated errors.

4.2 Backend: System Services

The *system interface* (Fig. 3) bridges the user interface and the NOW and also interfaces between the networked workstations. Visper's system interface uses two reliable communication techniques: point-to-point (TCP) and Intranet multicast (iBus) [31]. Fig. 5 shows the architecture and the main subsystems.

The backend provides the services that are used to run programs, generate debugging and runtime data, control the backend itself, and support fault tolerance. They are described in Sections 5, 6, and 7. The *membership service* is the database of available resources and detector of faulty hosts. The *resource management* provides runtime allocation of hosts. Each host participating in a Visper environment must run a *Visper daemon*. Each daemon forks one *worker* per session to run programs consisting of *remote threads (RT)* and maintains workers' state (e.g., active, dead). A daemon, with its set of workers, forms a federated system. As the basic building blocks in Visper programming, remote threads will be further discussed in Section 5.1. Fig. 5 illustrates that a worker can run multiple remote threads. The separation of the Visper daemon, worker, and resource manager into dedicated JVM processes facilitates different security policies and better resilience to malicious code. It also promotes separation of concerns, which enforces locality of different kinds of information in the programs, making them easier to understand, write, maintain, and modify.

To minimize the program start-up cost, each session maintains its own set of active workers. To allow location transparent programming and more efficient loading of Java class files, the user can define multiple loading points, and different access modes (e.g., http://..., file://...). This approach also eliminates a potential bottleneck caused by a single file server servicing many concurrent requests. The defined access path is valid across a whole network, rather

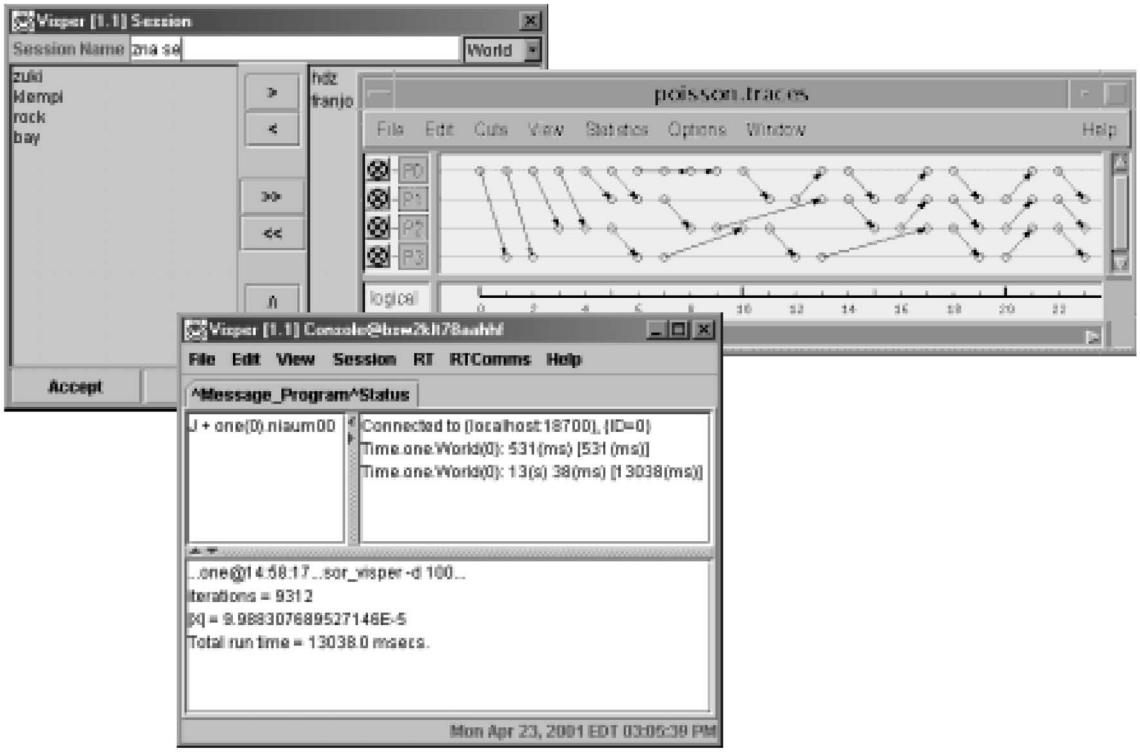


Fig. 4. Frontend components.

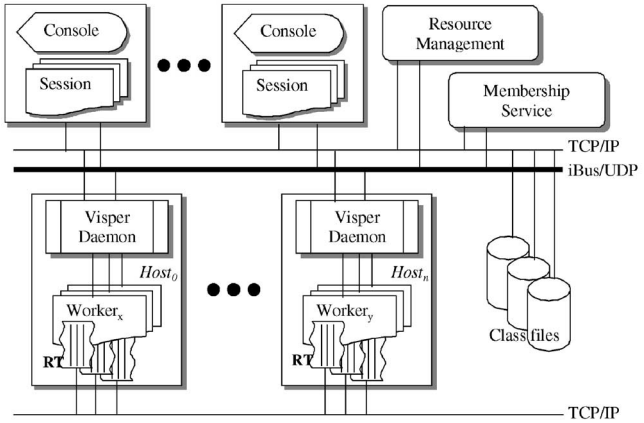


Fig. 5. System architecture.

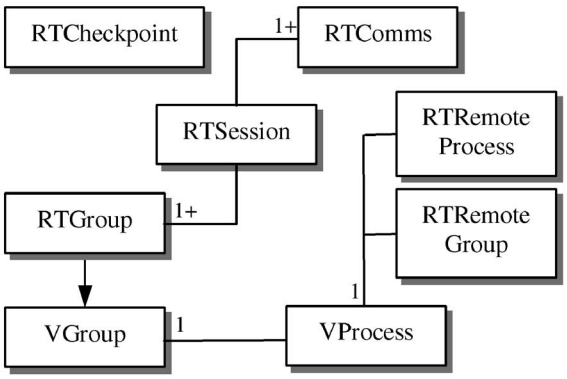


Fig. 6. Visper API classes.

than being local to a host like the *CLASSPATH* environment variable in Java.

5 Parallel Programming Support

From the programmer's perspective, Visper has a simple API, as shown in Fig. 6, that enables interaction, communication, and other activities between distributed objects. In the figure, the arrow line designates inheritance and the solid lines represent association. The system-level classes are prefixed with a *V* (for Visper) and the user-level classes with a *RT* (for remote thread). Most RT classes are just wrappers for the corresponding V superclasses. They abstract away implementation issues by hiding the internal package structure from the programmer and providing a simple and coherent API as a single package called *visper.rt*.

The execution model in Visper is based upon groups represented by *VGroup* and *RTGroup*. All computing resources are organized and accessed on a group basis. For example, the user first defines a session that represents a group of hosts. When instantiating a group, the system creates one process per worker in that group. At the API level, the programming primitives, such as *RTRemoteProcess* and *RTRemoteGroup*, transform the notion of a group of hosts into a group of processes on which a program that consists of remote threads runs. Processes are created, controlled, and populated by remote threads only within the context of a group to which they belong, i.e., no process exists outside a group boundary. Consequently, groups provide a communication scope for their member processes since one process refers to another as being a member of a particular group.

- *RTSession* implements an interface to the session abstraction. The role of a session is twofold. From the user's perspective, a session provides the means to control the environment. From the programmer's perspective, an RTSession provides the means to interact with the user and the environment. An RTSession contains information about the current session configuration and status, and keeps track of all the resources allocated by a program. For each execution of a program, there is only one session object.
- *RTGroup* represents the abstraction for allocation and ordering of processes. In Visper, each group is unique within its session, and represents the smallest unit of process organization.
- *RTRemoteProcess* and *RTRemoteGroup* provide methods to spawn and control one or more remote threads based upon a process group. *VProcess* is a system class that generates messages to control processes.
- *RTComms* implements communication and synchronization primitives as defined by the MPI standard. The supported primitives include point-to-point and collective communications that can behave as synchronous or asynchronous.
- *RTCheckpoint* implements the checkpointing and recovery mechanism to be explained in Section 6.

5.1 Remote Threads

In Visper, a remote thread is the unit of computation that encapsulates behavior and a resource. As such, it is the semantics object of an application. Remote threads can be downloaded or migrated, on demand, to where they have been scheduled to run. They are protocol and platform independent and they dynamically extend the functionality available at the remote hosts. Remote threads have the Java advantage: memory access violations and typing violations are not possible, so that faulty remote threads will not crash processes as they do in most native language environments. Any class can become a remote thread simply by implementing the *RTRunnable* interface:

```
public interface RTRunnable extends
java.io.Serializable {
    public void Run(String[] args,
    RTThreadGroup rttg,RTSession rts);
}
```

The *Run* method defines a remote thread body with a sequence of actions executed by a thread. The method takes three arguments. The first argument contains input arguments provided by the user. The second argument is used in those cases when program uses Java threads. All the spawned Java threads should belong to that thread group informing the system to maintain the current remote thread

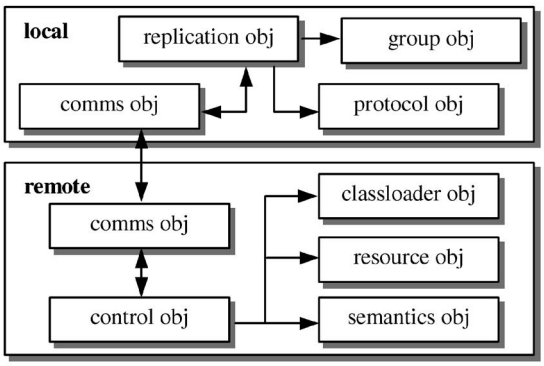


Fig. 7. Remote process architecture.

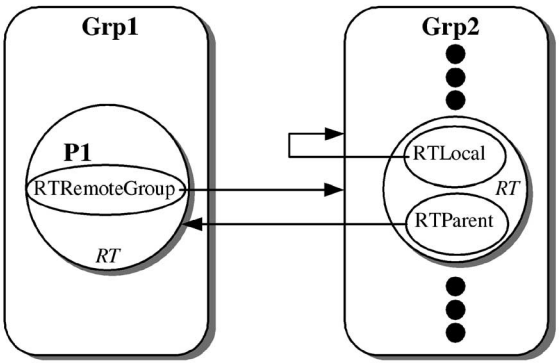


Fig. 8. Local and parent groups.

state before these Java threads *join*. The third argument represents a session.

This solution promotes code reuse and inheriting from a remote thread class is thus unnecessary. Further, it does not add any other service nor imply any programming model to the implementing class, except for turning it into a program ready for execution. All remote threads are known to the system, while all other objects used by a remote thread are local and known only to that thread.

There are four states, *fork, blocked, runnable*, and *dead*, in a remote thread lifecycle:

- The initial remote thread state is called *fork*. The system creates and attaches a thread to a process and sets up an environment in which the thread will run.
- Upon creation and initialization of a remote thread, the system starts the thread by invoking its Run method. The start event triggers this state transition by changing the state from fork to runnable.
- A runnable remote thread becomes *blocked* by either participating in an I/O operation or being explicitly *suspend*ed. When the I/O operation is completed, or passes a resume message, the thread becomes runnable again.
- When *Run* terminates, the remote thread state changes to *dead*. Sending a stop event to a thread can also trigger a state transition to a dead state. A remote thread termination activates garbage collection so that the resources are released and the thread becomes unreachable. The system also removes the Java threads spawned by the terminated remote thread.

5.2 Groups

A group is an ordered set of hosts (i.e., resources). Groups can be *static* (i.e., defined at compile time) or *dynamic* (e.g., allocated by a resource manager) and may have a virtual topology. They are unique; within a session, no two groups can have the same name. Groups are system-wide objects; if a host crashes, all the groups that reference that host are notified. An RTGroup exports its size and name and process ID. By default, each session has a group called *RTWorld* that comprises all the hosts in the session, allocated either statically or dynamically.

Visper provides two more groups, *RTParent* and *RTLocal* (Fig. 8), to simplify programming with RTRemoteProcess and RTRemoteGroup constructs. Both classes represent virtual groups since they do not allocate resources. The idea is to have a programming model that does not depend on a specific naming convention or forces the programmer to pass configuration information when new processes are allocated. An RTParent includes only one member that has created the remote process or remote group of processes. An RTLocal is an alias for the actual group that a process or group of processes belongs to.

5.3 Remote Processes

Remote threads are physically distributed; meaning that the program state might be partitioned or replicated across multiple hosts at the same time. However, programmers need not be concerned with this, given primitives that encapsulate all implementation aspects, such as control and communication protocols, replication strategies, and migration. Two API classes, namely, RTRemoteProcess and RTRemoteGroup (Fig. 6), implement the interface that allows thread groups of different cardinality being handled transparently and uniformly as a single entity.

The remote process architecture can be observed as a two-part composition that consists of local and remote objects (Fig. 7). The local objects are directly used by instances of RTRemoteProcess and RTRemoteGroup (i.e., *replication* object in Fig. 7) classes to spawn asynchronously new processes on remote hosts that will run remote threads. A process in Visper has only one remote thread of control so that there is a one-to-one mapping between remote threads and processes. The *group* object (i.e., instances of RTGroup) instructs the replication object where to spawn new processes. The *protocol* object (VProcess in Fig. 6) creates control protocol messages. It is implemented following the factory method pattern [16]. The *comms* object communicates control protocol messages to the remote hosts and acts as a placeholder for different networking protocols and protocol implementations. Depending on the problem size, it chooses between point-to-point and multicast communication channels being implemented by underlying communication services and accessed through a uniform interface.

At the remote side, the *control* object accepts control protocol messages and translates them into actions. For

example, when spawning a new process, it first instructs the *classloader* to load a new *semantics* object class and then completes the scenario by allocating a *resource* that places the semantics object into a Java thread. The control object maintains and can be queried about the state of the thread it controls.

No object, either local or remote, in the described flow of control makes any assumptions about the state or is aware of the nature of the semantics object. Therefore, we can define standard interfaces that are not affected by different policies. When creating an RTRemoteGroup, the programmer defines a remote thread class, input arguments, and a group in which to spawn new threads. When creating an RTRemoteProcess as coded below, the programmer also has to define a host (i.e., *process ID*) relative to that group on which to spawn a new remote thread.

```
public class RTRemoteProcess implements
java.io.Serializable {
    public RTRemoteProcess(Class cls,String[]
    args,RTGroup grp,int processID);
    public Object Invoke(java.lang.reflect.
    Method method,Object[] args);
    public Object Invoke(String method,
    Object[] args);
    public boolean IsAlive();
    public boolean Join();
    public boolean Migrate(int processID);
    public boolean Resume();
    boolean Start();
    public boolean Stop();
    public boolean Suspend();
}
```

An object having a reference to a remote process or group can control the process or group by invoking any of the interface methods, e.g., invoking *IsAlive()* to test if that remote thread or remote thread group is alive (either active or suspended). *Join* provides explicit synchronization with a remote thread termination. It blocks until a thread or a group of threads complete their execution, i.e., exit Run. *Invoke* enables method invocation on remote threads. The method invocation mechanism is based on the Java Reflection API [39], which is an example of structural reflection that gives us an insight into structural aspects of the classes and objects in the current JVM. A more transparent model of method invocation requires a preprocessor similar to that in Java and RMI or CORBA [40]. It, nevertheless, allows invocation of methods on a remote thread, as well as on a group of remote threads. Also, any process that refers to another process may invoke methods on it. The console implements a subset of the control protocol to control programs on a session basis (e.g., *Stop*, *Resume*). In fact, the console is the master that spawns a session worth of workers that belong to a group called *World*.

5.4 Message-Passing Primitives

At the application level, Visper provides the RTComms (a TCP/IP-based communication class) to enable direct process-to-process and collective communications (in blocking/nonblocking and raw/trace modes). In its layout, RTComms follows the MPI standard, while its implementation is specifically designed for our OO framework, where remote threads communicate by exchanging serializable objects.

RTComms hides the complexity of physical addressing and communication from the programmer, such as establishing communication channels and handling of network exceptions. Messages can be sent within a group or among different groups. The synchronization capabilities are represented by the barrier method. For brevity, we list below some of the implemented methods.

```
public class RTComms {
    public Object[] AllGather(RTDataGroup dg,
    Object data);
    public Object[] AllGather(RTDataGroupInter
    dg,Object data);
    public Object[] AllReduce(RTDataGroupInter
    dg,Object data,RTDataOp oper);
    public Object[] AllToAll(RTDataGroup
    dg,Object data);
    public boolean Barrier(RTDataBarrier db);
    public Object Bcast(RTDataBroadcast db,
    Object data);
    public boolean Cancel(RTDataSend ds);
    public Object[] Gather(RTDataGroup dg,
    Object data);
    public boolean Probe(RTDataSend ds);
    public boolean ProbeAsync(RTDataRecv dr);
    public Object Recv(RTDataRecv dr);
    public RTDataRecv RecvAsync(RTDataRecv
    dr);
    public Object Reduce(RTDataGroup dg,Object
    data,RTDataOp oper);
    public boolean Send(RTDataInter dr,Object
    data);
    public boolean Send(RTDataSend ds,Object
    data);
    public RTDataSend SendAsync(RTDataSend
    ds,Object data);
    public Object SendRecv(RTDataSendRecv
    dsr,Object data);
    public Object Test(RTDataRecv dr);
    public Object[] WaitAll(RTDataRecv[] dr);
    public boolean[]WaitSome(RTDataSend[] ds);
}
```

All the methods follow the same pattern, where first arguments represent an intelligent message envelope that knows how to match itself against a destination address, matching criteria, and communication primitive, and second argument represents a content. Similar to MPI, the programmer refers to transparent process identifiers, rather than network names and ports. However, RTComms is an object-based communication library, with no direct support for native data types. The benefit of this approach is that RTComms is not domain-specific and programmers do not have to concern themselves with data marshaling. The drawback is that the programmer must use a wrapper object when communicating native types.

5.5 Master-Worker Example

The following example makes use of some of the concepts described so far. It implements a master-worker scenario, where process *P1* (i.e., master) in *Grp1* creates a group called *Grp2* and populates it with remote threads (Fig. 8). The RTRemoteGroup object refers to the remote threads (i.e., workers) spawned by *P1*. After the workers from *Grp2* perform some calculations, *P1* collects the results via an intergroup gather.

The following two program excerpts implement the master part:

```
RTGroup Grp2 = new RTGroup("Workers",rts); //
create new group in this session
RTRemoteGroup rg = new RTRemoteGroup(Worker.
class,null,Grp2);
Object[] obj = comms.Gather(new RTDataGroup
Inter(10,P1,Grp1,Grp2),null);
```

and the worker part:

```
public class Worker implements RTRunnable {
  public void Run(String[] args,RTThreadGroup
  rttg,RTSession rts) {
    Object result = ...; // calculate result
    RTGroup parent = new RTParent(rts); // get
    parent, i.e., P1 of Grp1
    RTGroup local = new RTLocal(rts); // get
    local group, i.e., Grp2
    comms.Gather(new RTDataGroupInter(10,0,
    parent,local),result);
  }
}
```

The *Gather* primitive reads as: Collect all the results with tag *10* from *Grp2* at process *P1* of *Grp1*. At the worker, the system automatically maps group *local* to *Grp2* and process *0* of group *parent* to process *P1* of *Grp1*.

6 Fault Tolerance

While a NOW offers additional processing power, it also introduces new problems, including a possible failure of one or more hosts. As the number of hosts in a network increases, the chance of one of them failing during a computation increases exponentially. A fault tolerance mechanism is vital in a network environment because it allows a failed application to be restarted from the most recent saved state, thus reducing rerun time by skipping over the already computed operations. A *checkpoint* represents the saved state of a process. The procedure of restarting from a previously checkpointed state is called *rollback-recovery* that searches for a *consistent* system state [4] among the generated checkpoints.

6.1 The Checkpointing Model

In a network where programs execute asynchronously to achieve better performance, a checkpointing mechanism must follow the execution model to be least intrusive. On the other hand, the system should aim at generating a consistent preserved state, thus minimizing the network overhead and the amount of saved and potentially useless data.

Unlike other models, the Visper checkpoint model is a two-phase commit that combines merits of two different approaches. In the *uncoordinated* checkpointing protocol, each process in a message-passing application takes its checkpoints independently [11]. In the *coordinated* protocol, the processes in a system coordinate their checkpoints to form a system wide consistent state.

The model involves workers and parent Visper daemons (Fig. 5). From the worker's perspective, the checkpoint policy is optimistic (i.e., uncoordinated) because checkpoints are created independently. From the daemon's perspective, the policy is coordinated to simplify the recovery procedure and eliminate the domino effect [29] that can lead to an unbound rollback. To minimize the cost in time, the uncoordinated phase is synchronous and unbuffered since the mechanism does not make or maintain in-worker copies of checkpointed objects. The checkpointing mechanism relies on the Java Object Serialization and stream compression filters, which require that all objects registered with the fault tolerance mechanism are serializable. These objects are passed locally as a byte array, through a socket, to the parent daemon that implements the coordinated checkpointing policy. Each consistent checkpoint is identified by appending a monotonically increasing consistent checkpoint number (CCN) [10] to the checkpoint file name. The protocol proceeds as follows:

- A remote thread invokes a commit on a checkpoint that informs the parent daemon to create a new checkpoint. Daemon may either buffer in memory or save the checkpoint to a file, depending on the data size and caching policy. Then, it broadcasts a marker message with a CCN to inform other daemons about the tentative checkpoint.
- Based on session and group information, the receiving daemons then check if they should participate in the checkpoint. When all the participating daemons receive all marker messages from each other, a tentative checkpoint is qualified to become permanent. The data buffered in memory are saved to a file. If in the process of forming a new consistent checkpoint one or more hosts crash, the process is aborted by the membership service.
- The coordinator, always represented by the process whose ID is equal to 0 relative to that process group, broadcasts a commit message for synchronization. A tentative checkpoint becomes permanent and the previous permanent checkpoint is discarded.

The coordinated second phase of checkpointing automatically removes the previous consistent checkpoint. Since the first phase of checkpointing is uncoordinated, each remote thread may generate multiple tentative checkpoints that have to be garbage-collected when the program terminates either successfully or unsuccessfully. Upon a successful termination, all checkpoints are garbage-collected. Unsuccessful termination collects only tentative checkpoints. When recovering, the system reconstructs a consistent program state from the saved files. If a host fails, a complete program, rather than just the failed process or processes, is restarted.

6.2 The API

The checkpointing API assumes that a process consists of a single address space and resides on a single host. Since the JVM does not allow direct program counter and stack manipulations, a high level checkpointing is implemented as an API class called *RTCheckpoint* as shown below.

```
public class RTCheckpoint {
  public RTCheckpoint(String name,RTSession
  rts);
  public boolean Commit(); // mark checkpoint
  as completed
  public boolean Initialize(); // initialize
  checkpoint
  public boolean Recover(); // get last
  consistent program checkpoint
  public boolean RecoverLast(); // get last
  process checkpoint
  public Object Read(String name);
  public boolean Write(String name,
  Serializable obj);
}
```

Only the variables required to restore the program to the position of a given checkpoint need to be saved. By calling *Commit*, a checkpoint is marked as complete. Subsequent calls to *Write* form a new checkpoint. Checkpoint files use group names for consistency and all processes in a checkpointed group must invoke *Commit* on a checkpoint. Thus, the following code is incorrect:

```
if (processID % 2 == 0) checkpoint.Commit();
```

because it produces only 50 percent of the needed commit requests in creating a consistent checkpoint.

The current Visper prototype provides reliable communication channels but does not handle messages lost due to process failures. This implies that *in-transit* messages (those sent but not received at the failure instant) may cause a saved system state become inconsistent.

6.3 Recovery Through Thread Migration

In distributed programming, process migration is useful in allowing processes to be restarted from a known state in a different address space. Depending on the application, it can be used either to offload a host or to continue execution on a new host. In the *strong migration* scheme [18], the underlying system captures the complete process state and transfers it together with the code to a new location. At a new location, the process data and execution state are automatically restored. Due to the characteristics of the JVM, strong migration is not possible in Java without modifying the JVM or instrumenting the code [15]. In the *weak migration* scheme, only process data are transferred and, thus, the programmer must manually restore the preserved execution state upon migration.

As a part of the checkpointing mechanism, Visper implements weak migration of remote threads. Upon a request for evacuation, a complete remote thread is serialized. This operation captures all the attributes referenced by or contained in the remote thread. A remote thread can migrate only within its own group of instantiation and they cannot migrate across the boundary of a session. The migration mechanism can also help load balancing and speed up execution by migrating the threads to faster hosts that have become available.

7 Distributed Management

At the backend, software agents carry out Visper's system management in a distributed manner. The agents provide the backend components with the knowledge of their roles within the system and their behaviors. The agents that control various backend components can be dynamically reconfigured and manipulated for uninterrupted execution.

7.1 The Agent Model

A number of agent frameworks and tools in Java have been developed, such as IBM Aglets [1] and Voyager [26]. They are oriented towards building secure network-based applications for mobile agents, such as searching and managing data and other information. However, they do not facilitate coordination and cooperation among agents.

Visper is a multiagent environment that enables *static* agents to work autonomously in achieving their goals. Agents can also cooperate with other agents of the same or different class by exchanging high-level messages that incur lower communication costs. The Visper environment supports both the *contract net* and the *specification sharing* methods of interoperation for an agent to request a service or to advertise its presence. With the contract net approach [6], agents in need of services distribute requests to other agents. The recipients evaluate those requests and submit bids to the originating agents who then decide which agents to award with contracts. With the specification sharing approach [17], agents supply other agents with information about their capabilities and needs. These agents can then use this information to coordinate their activities.

Visper allows the system programmer to concentrate on agent behaviors rather than low-level details. Visper defines a common and effective messaging protocol to enable heterogeneous agents engage in conversation.

At the lowest level, the Visper agent model can be described as:

- **Behavioral**, as defined by Watt [42], since agents act autonomously and without knowledge or ability to reason about another agent's goals, intentions, or beliefs.
- **Static** since generally agents do not travel beyond their points of installation.
- **Anonymous** since the system applies no naming scheme upon agents. Due to anonymity, these agents have limited knowledge of the community in which they operate. For direct communication, an agent acquires its identity from the host daemon.
- **Reactive** since Visper agents act upon stimulus by responding to the present state of the environment [12] and the events (i.e., messages) they receive.

Agents can form groups. A typical form of grouping is session. To relieve the implementation of a router or a name server, such as the Agent Name Server (ANS) [22], Visper uses subject-based and multicast communications.

TABLE 1
Experimental Hardware

Vendor	Architecture	OS	RAM (MB)	CPU (MHz)
Sun	Ultra 2	Solaris 2.5	256	168 x 2
HP	A 9000/780	HPUX B.10.20	512	180
Compaq (PC1)	Pentium II	NT 4.00.1381	64	200
Micron (PC2)	Pentium II	NT 4.00.1381	384	400 x 2
Dell (PC3)	Pentium III	Win 2000 Prof.	256	500

Multicasting facilitates specification sharing and enables communication without prior knowledge of the other agents' whereabouts. Once they have decided with whom to communicate, they can establish a direct and dedicated communication channel.

Messages can be either passive or active. A passive message is an object that is hardcoded into the system. An active message contains not only data but also a behavior and executes in its own thread. For example, active messages are used in console-to-system communication to enhance the built-in system functionality without modifying the agent layer.

The above agent model serves as an experiment to demonstrate its feasibility in the Visper framework. The nature of agents may be changed to suit different purposes.

7.2 The API

The API of Visper agents consists of two base classes: *VAgent* and *VMessage*. VAgent is abstract and must be extended to create application-specific agent classes. Its interface defines three methods.

```
public abstract class VAgent implements
java.io.Serializable {
   public abstract void Decode(VMessage msg);
   public void Finalize();
   public void Initialize();
}
```

The *Initialize* method performs the initialization and gets invoked by a host daemon before the agent is attached to its message queue. After being attached, the agent is ready to process messages. Each time a new message arrives, *Decode* gets invoked with a message content passed in. Before an agent is removed from the system, *Finalize* gets called to release allocated resources.

VMessage is the base class for all messages used by the framework. Each message is uniquely identified by a tag and information about its origin. Using objects rather than interpretive messages simplifies message handling and eliminates inconsistencies and arbitrary notational variations.

Visper defines multiple agents. For example, the *VAgentDaemon* class implements the Visper daemon configuration agent, while the *VAgentWorker* implements the worker configuration agent. Following the federated organization among the system components (Section 4.2), a daemon agent also acts as the facilitator [17] to worker agents. Not all daemons or workers are required to run agents of the same class, allowing specialization through inheritance, but they all must understand the basic vocabulary. The same is valid for agents that run in workers or resource managers.

8 Performance Results

The advanced features of Java as a programming language and portability of Java programs come at a cost. The JVM is a heavy process that takes a couple of seconds to start and initially ~4 MB of RAM. To improve the runtime performance of Java programs, recent releases of the JDK come with compilers that translate the Java bytecodes to native instructions before the program starts executing. The problem here is that the compilation needs to be fast, thus lacking time to perform optimizations. Further, the compiled code cannot be preserved across multiple runs and the same price in time and performance is paid upon each program invocation. Visper has employed a number of techniques to deal with these issues.

To minimize the program startup cost, the Visper session is persistent. Further, a worker that outlives a program termination caches the classes and generated native code for subsequent runs. These caching capabilities allow a fast transition from an idle to a running state, together with faster overall execution speed. The cached classes are stored in a customized class loader and can be updated without restarting the worker. However, to avoid the risk of conflicts due to possible cache inconsistencies, when a single class is changed, the whole cache is dumped and reloaded.

To evaluate the performance of the Visper prototype, a number of computational benchmarks have been developed and executed in the environment. The performances for sequential execution, checkpointing, the speedups of parallel execution with multiple hosts, and the caching effect on multiple runs have been evaluated using several benchmark programs. This section reports three representative benchmarks and their results.

The experimental environment consists of hardware as detailed in Table 1. All the computers are connected with a 10 Mbps Ethernet. The network is partitioned in two subdomains: one for Unix workstations and one for PCs. Since there are three different architectures, three different versions of Java were used. A comparison of Visper to native systems was reported earlier [35].

TABLE 2
Magic Square (ms)

HW	Visper		Java
	1st run	2nd run	
PC2	1,016	328	1,063
PC1	3,174	1,152	2,994
HP	4,276	2,679	4,170

8.1 The Jama Benchmark

To test basic sequential performance of Visper, we used Jama [21], a standard matrix class for Java, to compare the performance of a standard JVM that does not cache classes across multiple runs to that of Visper. The difference between the Visper worker and the JVM is that Visper adds a layer of abstraction on top of JVM. A worker, as a Java program, runs in a JVM and each remote thread runs in a Java thread scheduled by the JVM.

The Magic Square program in Jama was converted into a Visper remote thread by implementing the RTRunnable interface instead of *main*. The results presented in Table 2 do not include the time to load (and initially just-in-time compile) the bytecodes. Since JVM does not cache classes across multiple runs, each program execution is equal to a 1st run in Visper. Therefore, the JVM results use only one column. The results show that a significant improvement in performance can be achieved, even when the program size and class structure are not complex.

8.2 Monte Carlo Computation of π

The Monte Carlo method of calculating π generates points and calculates how many points are in a circle, and how many are outside [20]. The testing of these points is highly parallelizable that compensates for the diversity in speed of our hardware. (Fig. 2 shows the program's visual form.)

The total number of iterations performed at each run was 2,000,000. The results do not include the time for an initial installation of remote threads, but only the total remote-thread execution time. The results show that a distinction can be made between a first run (*1st run* in Fig. 9) and subsequent runs (*2nd run)*, with or without just-in-time (JIT) compilation. This is due to the behavior of the Java class loader that resolves classes on demand, rather than at startup, and loads them one by one. In Visper, each worker loads program classes only once and they remain cached for subsequent runs. Since class loading is done via point-to-point communications, the caching of the classes yields better performance, even in simple cases (Table 3). Another approach to minimize the 1st run cost could be taken by preloading classes. However, this approach is useful when resources are not allocated dynamically at runtime.

In Fig. 9, *1st run* and *1st run speedup* represent the time and speedup, respectively, when all the workers were running the program for the first time and no classes were cached locally. *2nd run* and *2nd run speedup* represent the results with cached classes. *1st run* represents the actual values, while *2nd run* represents the average value over nine runs. With the increasing number of hosts, the difference between the *1st run* and the *2nd run* times increases, even though the program uses only 14 classes.

As a reference, the *1 host* line represents the time required for one worker to perform *2,000,000/hosts* iterations and send the result back locally. We use it to compare the message-passing overhead caused by the network to the time of a message passed locally to the host. As expected, the speedup decreased as new hosts were added to the system because the process that generated random numbers got more requests. The flat section between hosts 5 and 6 is due to a small difference in the number of performed iterations as more processes join in, which additionally flattens out the curve.

In summary, the performance evaluation has confirmed the following:

- Given a reasonable coarse-grain parallelism, where the time spent computing is much greater than the time spent communicating, Visper shows speedups when increasing the number of participating hosts.
- The Visper implementation in Java does not suit applications with fine-grain parallelism, due to a

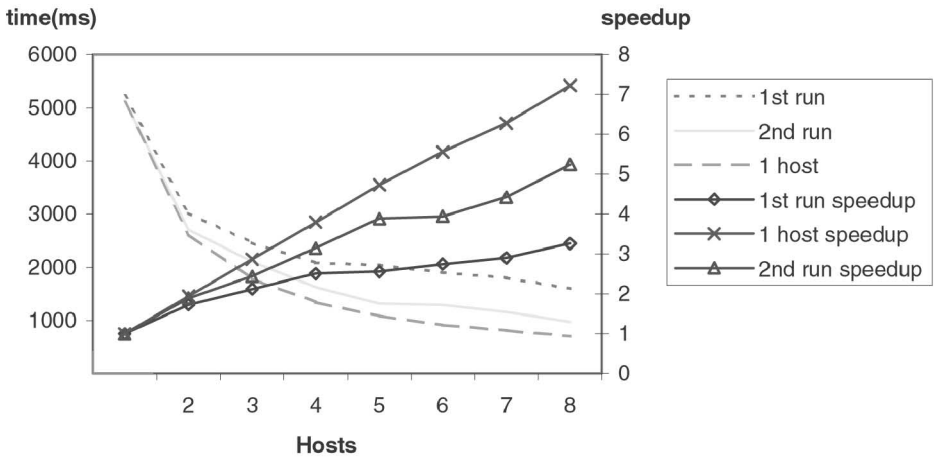


Fig. 9. π Calculation performance data.

TABLE 3
JIT and Class Caching (ms)

Iterations	1st run	2nd run	3rd run
1,000,000	951	513	460
500,000	757	250	261

relatively high communication cost in the range of milliseconds [34].
- The behavior of the Visper prototype in running these benchmark programs is predictable and indeed produced results as expected, in contrast to some related systems (Section 9), which produced unpredictable performances [34].
- Caching can significantly improve computational time, even for simple programs with only one worker executing. It also gives the just-in-time compiler more time to optimize code.

8.3 Checkpointing

Here, we detail performance results of the checkpointing mechanism, collected on *PC3* (Table 1). The presented results include the cost of checkpointing a two-dimensional double precision array of different sizes. In each array, the first element was always set to 10,000 and the other elements incremented by one, consecutively. The values in Table 4 include the time:

- to compress a wrapper object that holds a reference to the array and convert it to bytes,
- to serialize the bytes together with some additional information (3 objects), and
- to synchronously transport the serialized data to a parent daemon.

We can observe that the time to push the array out of the worker is proportional to the array size. The whole checkpointing procedure is efficient and does not suffer from networking overheads. The generated file is small, due to a compression ratio of ~3.5.

The second test investigates the cost of checkpointing when performing successive over relaxation (SOR) on a Poisson system. This is a JPVM [13] program modified for Visper that uses the red-black ordering of the unknowns. The results presented in Table 5 include the time to save a square matrix of doubles, by generating seven checkpoint files on a mounted file system. For comparison, we also list the time for the program to run without checkpointing. As expected, checkpointing exhibits very small impact on the total execution time. Therefore, the execution time is more susceptible to slowdowns coming from the outside (e.g., 180×180 test case). This confirms the results from Table 4, even though the generated files are much larger in this case.

TABLE 4
Two-Dimensional Arrays of Doubles

Array Size	File (kB)	Time (ms)
100x100	23	142
150x150	51	328
180x180	74	479
200x200	91	600
250x250	141	925
300x300	196	1,285
500x500	528	3,223
1,000x1,000	2,030	13,782

9 Related Works

A standalone parallel programming environment can be built either by extending the Java language with new keywords [28] or by providing a pure Java API for parallel programming. Here, only pure Java systems are of interest and among which we classify Visper since new keywords require nonstandard Java components. None of the related systems supports visual parallel program development, with fault tolerance and process migration being generally weak.

JPVM [13] is an attempt at PVM in Java, but misses some important features, e.g., dynamic process group and group broadcast. The process creation in JPVM is just a mechanism, while, in Visper, it is a programming primitive that is also used for coordination and control. IceT [19] also

TABLE 5
SOR on Poisson

Matrix Size	File (kB)	Iterations	No of CP	Time (ms)
150x150	135	21,774	0	86,848
			7	93,648
180x180	207	31,844	0	218,374
			7	217,232
200x200	262	39,699	0	378,164
			7	379,713
250x250	423	63,325	0	1,279,796
			7	1,281,557

TABLE 6
Comparison

Feature	DOGMA	JPVM	IceT	Ninflet	Visper
Scalability	N	N		Y	Y
Scheduling	Y	N	N	Y	Y
Load Balancing	N	N	N	Y	N
Fault Tolerance	N	N	N	Y	Y
Extensibility	N	N	N	N	Y
Interoperability	N	N	Y		N
Migration	N	N	Y	Y	Y
Data Scheme	Y	N	N	N	N
Clusters	Y	N	N	Y	N

follows PVM, being enhanced with classes for collaborative work of multiple users.

DOGMA [7] runs parallel programs on a network of workstations and supercomputers (IBM SP/2) based on the MPI model. The communication library, called MPIJ, implements MPI completely in Java. Unlike the RTComms in Visper, the MPIJ implementation of MPI is based on the MPI C++ bindings *as much as possible*. There is no support for passing objects, as it focuses on native types and efficiency. DOGMA nodes cache classes locally, but does not allow updates without manually removing and restarting all the nodes, which makes development and debugging very difficult.

Ninflet [41] is a Java based global computing environment that builds on the experience acquired by the Ninf system. A ninflet is a schedulable client program that executes on the Ninflet system. Ninflets interact by invoking methods based on the RMI.

Table 6 summarizes the important features supported in these systems. It is based on the list of objectives that DOGMA seeks to meet. Ninflet is included primarily as a fairly complete distributed programming system because it lacks message-passing primitives.

Scalability in DOGMA and JPVM is questionable since both systems use persistent communication channels to improve performance. While the performance of MPIJ justifies the approach, that of JPVM is erratic [34]. In pure Java, load balancing is an open issue due to a rather limited interface to the OS. Interoperability requires resolving the initialization of the native PVM or MPI virtual machine from Java, vice-versa, and data representation. As a proof of concept, IceT managed to soft-install C-based MPI processes on remote environments and dynamically install FORTRAN-base PVM. The support for clusters in DOGMA is part of its hierarchical topology.

While all the mentioned systems have been tied to the style of legacy systems, Visper is designed to allow different programming models and applications being executed within the same extensible framework. Rather than making Java behave like legacy systems as much as possible, we have decided to introduce to Java those techniques in the most natural way possible. This is particularly important due to the lack of pointer arithmetic in Java and because Java types and arrays do not map naturally to native types. Therefore, the support for parallel programming in Visper is just a service, rather than a goal.

In Visper, the network is a flat resource organized into sessions. However, all the hosts that form a session can be manually ordered in a session dialog (Fig. 4) to support the MPI model or if so desired. The support for remote thread migration in Visper is useful when a remote thread initiates and conducts the migration process by itself. We find this kind of behavior in mobile agents. For example, the user first defines a session with a list of hosts and then starts a remote thread that visits all these hosts and performs tasks locally. Thus, the order in which hosts appear in a session is used to externally define an itinerary the agent follows. Externally initiated migration as in Ninflet is not fully supported in the current prototype of Visper.

10 Conclusion

Visper is conceived as a generic, distributed programming environment that identifies the services that are common among different programming models. These services have been implemented in such a way that they can be used consistently across multiple models. Visper is an object-oriented programming environment that is open, secure, and fault tolerant, with static and dynamic resource management. It allows remote execution of Java programs in the form of remote threads, by transforming a network of computers into a metacomputer called session. Remote threads are autonomous, interacting computing elements that encapsulate data and behavior. They can be dynamically instantiated, configured, and controlled, thus providing flexibility in organizing their activities. The session in Visper is persistent, meaning that it survives a program termination and program neutral meaning that multiple programs can run on the same session, one at a time. We believe that the concept of remote thread and session in Visper is more generic than similar approaches, such as Aglets [1] and Ninflet [41]. Each of the latter approaches has identified a specific (agent) type that is self-sufficient and, therefore, more constrained.

As a parallel programming environment, Visper offers a set of tools by which the programmer can compose, run, and test parallel programs within the same visual formalism. Further, the proposed model combines the standard practices and techniques pioneered by MPI and PVM with the presented new ideas and features offered by Java. A parallel program executes as a group or groups of asynchronous remote threads that communicate via message passing. Visper decouples the system services and system configuration, such as process and group management, from the message-passing API. In communications, following the message-passing model, passive objects (data) are passed as parameters to the methods of the Visper communication class called RTComms. The communication primitives represented by RTComms follow the primitives of the MPI standard. They provide synchronous and asynchronous modes of communication among processes. The programmer can choose between point-to-point and collective communications. The implementation does not obscure the usage of other communication mechanisms, such as the Java RMI [38], if appropriate.

APPENDIX A
PROGRAM EXAMPLE

The following is a simple example that shows how to use the checkpointing mechanism in Visper to achieve fault tolerance. The program saves the index periodically after 10 iterations.

```
import visper.rt.*;
public class SendReceiveMPI implements
RTRunnable {
  public void Run(String[] args,RTThreadGroup
  rttg,RTSession rts)
  {
     int i = 1;
     RTCheckpoint rtc
     = new RTCheckpoint("cp",rts)
     RTComms comms = new RTComms(rts);
     // use default communications
     RTWorld world = new RTWorld(rts);
     // use default group

     if (rts.Restarted() && rtc.Recover())
       i = ((Integer)rtc.Read
       ("index")).intValue();
     rtc.Initialize();

     for (;i < 101;i++) {
  if (world.HostID() == 0) {
            RTDataSend ds =
            new RTDataSend(10,1,world);
            comms.Send(ds,new Integer(i));
            // blocking send
       } else if (rts.HostID(world) == 1) {
            RTDataRecv dr =
            new RTDataRecv(10,0,world);
            Integer msg = (Integer)comms.Recv
            (dr); // blocking receive
            rts.Out(msg.toString());
            // send to console
       }
       if (i % 10 == 0) {
            rtc.Write("index",new
            Integer(i));
            rtc.Commit();
       }
     }
  }
}
```

ACKNOWLEDGMENTS

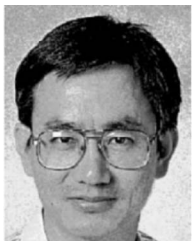
The authors would like to thank the anonymous reviewers for their insightful and constructive comments that have helped them to improve the final presentation of the paper.

REFERENCES

[1] "Aglets: Programming Mobile Agents in Java,"http://www.trl.ibm.co.jp/aglets, 1998.
[2] J.C. Browne, S.I. Hyder, J.J. Dongarra, K. Moore, and P.W. Newton, "Visual Programming and Debugging for Parallel Computing," *IEEE Parallel and Distributed Technology,* pp. 75-83, 1995.
[3] M.M. Burnett, M.J. Baker, C. Bohus, P. Carlson, S. Yang, and P. van Zee, "Scaling up Visual Programming Languages," *Computer,* pp. 45-54, Mar. 1995.
[4] S.K. Chang, M.M. Burnett, S. Levialdi, K. Marriott, J. Pfeiffer, and S. Tanimoto, "The Future of Visual Languages," *Proc. 15th IEEE Symp. Visual Languages,* pp. 58-61, 1999.
[5] K.M. Chandy and L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems," *ACM Trans. Computer Systems,* vol. 3, no. 1, pp. 63-75, 1985.
[6] R. Davis and R.G. Smith, "Negotiation as a Metaphor for Distributed Problem Solving," *Artificial Intelligence,* vol. 20, no. 1, pp. 63-109, 1983.
[7] DOGMA: Distributed Object Group Metacomputing Architecture, http://ccc.cs.byu.edu/DOGMA, Sept. 1998.
[8] R.A. Duncan, "Survey of Parallel Computer Architectures," *Computer,* vol. 23, no. 2, pp. 5-16, Feb. 1990.
[9] P. Eades, W. Lai, K. Misue, and K. Sugiyama, "Preserving the Mental Map of a Diagram," *Proc. Compugraphics 91,* pp. 24-33, 1991.
[10] E.N. Elnozahy, D.B. Johnson, and W. Zwaenepoel, "The Performance of Consistent Checkpointing," *Proc. 11th Symp. Reliable Distributed Systems,* pp. 39-47, Oct. 1992.
[11] M. Elnozahy, L. Alvisi, Y.M. Wang, and D.B. Johnson, "A Survey of Rollback-Recovery Protocols in Message-Passing Systems," Technical Report CMU-CS-99-148, School of CS, Carnegie Mellon Univ., June 1999.
[12] J. Ferber, "Stimulating with Reactive Agents," *Many Agent Simulation and Artificial Life,* E. Hillebrand and J. Stender, eds., pp. 8-28, 1994.
[13] A. Ferrari, "JPVM: Network Parallel Computing in Java," *Concurrency: Practice and Experience,* vol. 10, nos. 11-13, pp. 985-992, 1998.
[14] M.J. Fischer, N.A. Lynch, and M.S. Paterson, "Impossibility of Distributed Consensus with One Faulty Process," *J. ACM,* vol. 32, no. 2, pp. 374-382, 1985.
[15] S. Fünfrocken, "Transparent Migration of Java-Based Mobile Agents," K. Rothermel, and F. Hohl, eds., *Proc. Second Int'l Workshop Mobile Agents, (MA '98),* 1998.
[16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns—Elements of Reusable Object-Oriented Software,* Reading, Mass.: Addison-Wesley, 1996.
[17] M.R. Genesereth and S.P. Ketchpel, "Software Agents," *Comm. ACM,* vol. 37, no. 7, pp. 48-53, 1994.
[18] C. Ghezzi and G. Vigna, "Mobile Code Paradigms and Technologies: A Case Study," K. Rothermel and R. Popescu-Zeletin, eds., *Proc. First Int'l Workshop Mobile Agents (MA '97),* 1997.

[19] P.A. Gray and V.S. Sunderam, "Metacomputing with the IceT System," *Int'l J. High Performance Computing Applications,* vol. 13, no. 3, pp. 241-252, 1999.
[20] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI, Portable Parallel Programming with the Message-Passing Interface.* The MIT Press, 1994.
[21] "Jama: A Java Matrix Package," http://math.nist.gov/javanumerics/jama, 1998.
[22] "JATLite Overview,"Computer Science Dept., Stanford Univ., http://java.stanford.edu/java_agent/html, 1997.
[23] D. Kranzlmüller, N. Stankovic, and J. Volkert, "Debugging Parallel Programs with Visual Patterns," *Proc. 15th IEEE Int'l Symp. Visual Languages, (VL '99),* pp. 180-181, Sept. 1999.
[24] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Comm. ACM,* vol. 21, no. 7, pp. 558-565, July 1978.
[25] Message Passing Interface Forum, *MPI-2: Extensions to the Message-Passing Interface,* http://www.mcs.anl.gov/mpi, July 1997.
[26] "ObjectSpace Voyager—The Agent ORB for Java," http://www.objectspace.com/voyager, 1997.
[27] Pasadena Working Group #7, "DRAFT: Message-Passing and Object-Oriented Programming," *Proc. Second Pasadena Workshop System Software and Tools for High Performance Computing Environments,* http://cesdis.gsfc.nasa.gov/, 1995.
[28] M. Philippsen and M. Zenger, "JavaParty—Transparent Remote Objects in Java," *Concurrency: Practice and Experience,* vol. 9, no. 11, pp. 1225-1242, 1997.
[29] B. Randell, "System Structure for Software Fault Tolerance," *IEEE Trans. Software Eng.,* vol. 1, no. 2, pp. 220-232, June 1975.
[30] R.D. Schlichting and F.B. Schneider, "Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems," *ACM Trans. Computer Systems,* vol. 1, no. 3, pp. 222-238, 1983.
[31] *SoftWired AG. Ibus Programmer's Manual,* version 0.5. http://www.softwired.ch/ibus.htm, Aug. 1998.
[32] N. Stankovic and K. Zhang, "Visual Programming for Message-Passing Systems," *Int'l J. Software Eng. and Knowledge Eng.,* vol. 9, no. 4, pp. 397-423, Aug. 1999.
[33] N. Stankovic and K. Zhang, "A Framework for Object-Oriented Metacomputing," *Proc. Third Int'l Symp. Computing in Object-Oriented Parallel Environments, (ISCOPE '99),* S. Matsuoka, R.R. Oldehoeft, and M. Tholburn, eds., pp. 71-76, Dec. 1999.
[34] N. Stankovic and K. Zhang, "An Evaluation of Java Implementations of Message-Passing," *Software: Practice and Experience,* vol. 30, no. 7, pp. 741-763, June 2000.
[35] N. Stankovic, "An Open Java System for SPMD Programming," *Concurrency: Practice and Experience,* vol. 12, no. 11, pp. 1051-1076, Sept. 2000.
[36] N. Stankovic, D. Kranzlmüller, and K. Zhang, "The PCG: An Empirical Study," *J. Visual Languages and Computing,* vol. 12, no. 2, pp. 203-216, Apr. 2001.
[37] J.M. Stone, "Debugging Concurrent Processes: A Case Study," *Proc. ACM SIGPLAN 1988 Programming Language Design and Implementation,* pp. 145-153, June 1988.
[38] *Java Remote Method Invocation Specification,* revision 1.42. Sun Microsystems, Inc., Oct. 1997.
[39] *Java Core Reflection, API and Specification.* Sun Microsystems, Inc., Feb. 1997.
[40] Sun Microsystems, Inc., "Java IDL," http://java.sun.com/, 1999.
[41] H. Takagi, S. Matsuoka, H. Nakada, S. Sekiguchi, M. Satoh, and U. Nagashima, "Ninflet: A Migratable Parallel Objects Framework Using Java," *Proc. ACM 1998 Workshop Java for High-Performance Network Computing,* pp. 151-159, Feb. 1998.
[42] S.N.K. Watt, "Artificial Societies and Psychological Agents," *Software Agents and Soft Computing,* H.S. Nwana, and N. Azarmi, eds., *Towards Enhancing Machine Intelligence, Concepts, and Applications,* pp. 27-41, 1997.
[43] K. Zhang, X. Ma, and T. Hintz, "The Role of Graphics in Parallel Program Development," *J. Visual Languages and Computing,* vol. 10, no. 3, pp. 215-243, 1999.

Nenad Stankovic received the BS degree in electrical engineering from the University of Zagreb, Croatia in 1983 and the MSc(Hons) and PhD degrees, both in computer science, from Macquarie University, Australia in 1997 and 2001, respectively. His research interests are in the areas of parallel programming environments, parallel and distributed systems' architecture, software agents, and artificial intelligence. Dr Stankovic has been working in industry on different projects in computing and currently holds a position at Nokia.



Kang Zhang received the BEng degree in computer engineering from the University of Electronic Science and Technology, China, in 1982, and the PhD degree from the University of Brighton, United Kingdom, in 1990. He is currently an associate professor at the University of Texas at Dallas. He has held positions as a lecturer and senior lecturer at Macquarie University, Sydney, Australia, a research assistant and SERC postdoctoral fellow at the University of Brighton, UK, and a software engineer at the East-China Research Institute of Computer Technology, Shanghai, China. Dr Zhang's research interests are in software visualization, parallel programming, visual programming, and Internet computing. Dr Zhang is a senior member of IEEE.