JavaParty – Transparent Remote Objects in Java

Michael Philippsen and Matthias Zenger
University of Karlsruhe, Germany
phlipp@ira.uka.de

Abstract

Java's threads offer appropriate means either for parallel programming of SMPs or as target constructs when compiling add-on features (e.g. forall constructs, automatic parallelization, etc.)

Unfortunately, Java does not provide elegant and straightforward mechanisms for parallel programming on distributed memory machines, like clusters of workstations.

JavaParty transparently adds remote objects to Java purely by declaration while avoiding disadvantages of explicit socket communication, the programming overhead of RMI, and many disadvantages of the message-passing approach in general. JavaParty is specifically targeted towards and implemented on clusters of workstations. It hence combines Java-like programming and the concepts of distributed shared memory in heterogeneous networks.

1 Introduction

Among the essential features that made Java [6] popular are (a) the availability of portable Internet communication APIs [7], like for example socket communication, and (b) the fact that threads and synchronization mechanisms are part of the language [14].

Java's threads allow for portable parallel programming within the bounds of shared memory; if the hardware offers several processors, (in theory) a threaded application can gain speed.[1] For wide area client-server applications, the communication libraries offer a good set of tools for implementing arbitrary communication protocols.

If the intended protocol can be reduced to method invocations on remote Java objects, then RMI [20] can be used. With RMI, Java objects that reside on different hosts can be used in a CORBA-like fashion, i.e., remote object references provided by a name server are bound to local variables. Then, methods can be called on those remote objects. RMI offers a rich set of exceptions to deal with network problems.

Therefore, Java offers good support for medium grain shared memory parallelism and for medium to coarse grain distributed parallel applications with restricted communication needs. However, Java does neither offer any significant support for cluster computing or workstation-based parallel computing nor does it address more involved or irregular communication needs.

Currently several projects are under way that achieve increased bandwidth, reduced latency, and better reliability within workstation clusters [1, 2, 21]. These projects will fill the gaps between desktop SMPs, parallel computers, and Internet-based meta computing.

A similar gap can be noticed on the software side. With improved cluster interconnection technology and growing cluster reliability and availability, Java's current mechanisms become insufficient: Explicit socket communication is too low-level for comfortable parallel programming. RMI is too restrictive and its overhead for dealing with network problems is too verbose for a cluster setting. Both sockets and RMI result in increased program size and thus reduce programmer productivity and code maintainability.

JavaParty fills this gap.[2] It extends Java as minimally and transparently as possible with a pre-processor and a run-time system for distributed parallel programming in heterogeneous workstation clusters.

Although in principle a JavaParty can include any number and type of workstations, high network latencies and low network speeds restrict the usefulness of traditional networks. Better runtimes can be achieved with special interconnection hardware.[3] Our current JavaParty implementation runs in a ParaStation network [21], where TCP/IP-like sockets deliver 15 MByte/s for small-packet point-to-point communication with 3.4 μs latency.

JavaParty is a two purpose platform. It serves as a programming environment for cluster applications and it is a basis for computer science research in optimization techniques to improve locality and reduce communication time. Several projects are under way that either use the JavaParty platform or improve it. The projects – some of which are supported by the DFG, Germany's national science foundation – include a data-intensive geophysical application (in cooperation with the Stanford Exploration Project [3]), a data mining project, real-time vehicle track-

[1] Currently some implementation restrictions apply: On several SMP platforms multiple threads are still scheduled to a single processor, and not all platforms implement pre-emptive scheduling.

[2] Think of JavaParty as a party of Java virtual machines cooperating on a common task.

[3] For the remainder of this paper, we consider workstation-based parallel computers as a specialization of workstation clusters.

ing in traffic scenes, and locality optimization and improvement of the underlying communication libraries.

In section 2 we discuss the features of JavaParty. Section 3 compares JavaParty code to code that uses the network API and to code that uses RMI. The fourth section presents the design and implementation of JavaParty in some detail. After a look at related work in section 5 we conclude this paper.

2 JavaParty

A multi-threaded Java program can easily be turned into a distributed JavaParty program by identifying those classes and threads that should be spread across the distributed environment. The programmer indicates this by a newly introduced class modifier `remote`. The new modifier is the only extension of Java. Since Java's threads are objects of a thread class, remote threads can be created as objects of a remote thread class. There is no need to significantly re-write or re-organize a given Java program nor does it grow in size.

JavaParty provides a shared address space, i.e., although objects of remote classes may reside on different machines their methods and variables (both non-static and static) can be accessed in the same way as in pure Java. Since JavaParty hides addressing and communication mechanisms from the user and handles network exceptions internally, no explicit communication protocol needs to be designed and implemented by the programmer.

The modifier `remote` is the only extension of the language. We would have loved to even avoid this extension, but there is no way to transform the basic JDK library classes into remote classes for the following reasons: We do not have the source code of the complete JDK; and even if we had, several classes have quite a lot of native code that is designed specifically for single processor implementations, e.g., `Thread`, `I/O`, `Runtime`, or `System`. Upgrading these classes to transparently implement remote semantics would require a major redesign of the class libraries and the Java runtime system. Moreover, JavaParty would no longer be available on all platforms, since it would differ from the standard JDK.

JavaParty is location transparent, i.e., the programmer does not need to map remote objects and remote threads to specific nodes of the network; compiler and runtime system deal with locality and communication optimization. Objects that are stored locally are used locally at the cost of a pointer indirection instead of expensive OS communication overhead.

To achieve location transparency, JavaParty offers distribution strategies that are used when new objects are created. Distribution strategies are implemented in the runtime system using the "strategy" design pattern [5]; they can thus be selected and changed at runtime. Compiler analysis (or a well-informed programmer) can insert code that directs the strategy's placement decision.

In addition to distribution strategies for object creation, the runtime system monitors the interaction of remote objects and the induced communication paths. If appropriate, the runtime system (or the well-informed programmer) schedules object migration to enhance locality. The runtime system is based on load-balancing and network partitioning algorithms. Currently replication is not considered, although it could easily be added, at least for static variables and methods.

3 JavaParty versus Sockets and RMI

For benchmarking purposes, we implemented some of the Salishan Problems [4] four times: in sequential Java, in Java with explicit socket communication, in Java with RMI, and in JavaParty. Problems include the ordered computation of all isomers of paraffin molecules (without repetition), a queue simulation of a doctor's office with multiple doctors, multiple patients and a potentially replicated receptionist, and Hamming's problem. To understand the discussion below, it is not necessary to have a deep understanding of the problems except that they are quite irregular and unpredictable and much different from typical numerical applications. Moreover, we implemented a multi-threaded version of a geophysical method, called Veltran or "normal moveout" in Java and in JavaParty.

Code size is the most significant difference between the four program versions.

		java	sockets	RMI	Java Party
`wc -l`	Salishan	1277	2086 63.3%	2123 66.2%	1277
	Veltran	967	–	–	967
`sdiff`	Salishan	0	992 77.7%	969 75.9%	28 2.2%
	Veltran	0	–	–	7 0.7%

Both the sequential Java programs and the JavaParty versions have the same number of lines, namely 1277 for the Salishan problems and 967 for Veltran. In contrast, code size increases by about 63.3% for the versions using sockets to a total of 2086 lines. It grows by 66.2% on average for the RMI versions (2123 lines).

The results are even more significant, if the edit distance is considered, i.e., if we count the number of lines that have been added, changed, or deleted, see `sdiff` lines of the above table.[4] These numbers more appropriately reflect the amount of work that had to be done to construct a distributed version from a threaded Java program.

Interestingly, although code sizes are quite similar, it took more than twice as long to get the socket versions right than we needed for the RMI versions.

[4] This information is computed by: `sdiff -sb file1.java file2.java | egrep -c '[<|>]'`

The JavaParty implementation of the Veltran operator on a four processor SGI PowerChallenge runs only about twice as long as the corresponding implementation in parallel Fortran.

Since our port of RMI to ParaStation sockets is still incomplete, we can only give qualitative performance results that have been measured on standard workstations. Since JavaParty is currently implemented on top of RMI (see section 4), we measure similar runtimes for the RMI versions and the JavaParty versions. However, where locality can be exploited, JavaParty easily outperforms the RMI versions. The socket versions are faster for smaller problem sizes, since RMI suffers from a significant startup time and marshaling/unmarshaling of arguments is more costly than in explicit socket protocols.

The following comparison focuses on program structure, programming experiences, and performance problems.

- **Program Structure and Remote Object Creation.** Because of the shared address space approach of JavaParty, neither artificial separations into client and server portions nor complex code for creation of remote objects is needed. This is quite different for socket and RMI versions.

  In general, socket and RMI programs follow the client/server approach. The programmer has to identify client portions and server portions and he has to explicitly allocate them to the underlying machines.

  Let us assume that a given threaded Java program can easily be cut into client and server portions. For the socket and the RMI versions, one then has to write at least two different programs, i.e., code for a server and for the clients. These programs have to be started manually or by an additional script on the various hosts. Separation into two or more programs causes a slight overhead in total code size.

  If the two portions are not obvious, use of sockets or RMI requires more work, since there is no straightforward way to create remote objects. For the socket solution, the communication protocol must be extended with commands that cause the recipient on a remote machine to create a new object on that machine by calling the appropriate constructor. Similar for RMI: on the remote machine an additional remote object is needed that offers a method that itself calls a constructor of the desired class.

- **Connection Setup.** JavaParty hides the addressing and communication mechanisms from the user, handles network exceptions internally, and thus keeps the code smaller than socket and RMI versions.

  To connect server and client processes, the socket and RMI programs must solve TCP/IP addressing tasks. They must access the IP name of their host machine, they must know about port numbers and about names used to register objects with the RMI name server.

  The following code is used at the client side to open the socket connection and to get hold of appropriate streams. It is left to the programmer to handle IOExceptions that might occur for example because of busy ports or race conditions that result when a server has not yet offered a connection.

```
DataInputStream is;
DataOutputStream os;
try {
  Socket MySocket = new Socket(server, port);
  is = new DataInputStream(MySocket.
           getInputStream());
  os = new DataOutputStream(MySocket.
           getOutputStream());
} catch (IOException e) {
  // ... what to do?
  // ... Try again? Involve the user?
}
```

  In the RMI versions, the server must be registered at the name server, called RMI registry, of its host. An RMI registry process must be running on each host that implements a remote object, i.e., a server process.

```
// create server
Server server = null;
try {
  server = new Server(...);
} catch (RemoteException e) {
  ...what to do?
}
// register server
try {
  InetAddress iaddr =
           InetAddress.getLocalHost();
  String url = "//"+iaddr.getHostName()+
           "/server";
  java.rmi.Naming.bind(url, server);
  server.work();
} catch (AlreadyBoundException e) {
  ... what to do?
} catch (MalformedURLException e) {
  ... what to do?
} catch (java.rmi.UnknownHostException e) {
  ... what to do?
} catch (java.net.UnknownHostException e) {
  ... what to do?
} catch (RemoteException e) {
  ... what to do?
}
```

  The server program needs similar code for termination, i.e., to remove the server object from the RMI registry. The client program needs similar code to get access to the server. Hence, the programmer has to deal with the exceptions mentioned above at least three times. Note that the above code is simplified with respect to host names.

Whereas these three points can be hidden behind a layer of abstraction, for method calls hiding is no longer possible without a complete re-engineering of given code (see bullet Communication/Invocation below).

Whereas `RemoteExceptions` do occur in wide area networks they are quite unlikely in clusters, unless caused by the socket or RMI model itself. A programmer who does not have to construct URLs and IP addresses will not run into errors and will not cause the corresponding exceptions.

Section 4 presents in detail how JavaParty achieves transparent connection setup.

- **Communication/Invocation.** In JavaParty, there is no need to design and implement the communication protocol by hand. JavaParty does not require to change method signatures and variables and hence neither causes major code re-working, nor does it require dealing with network exceptions.

  The socket versions cannot invoke methods of remote objects. Instead, the programmer has to design a communication protocol that must be implemented by two automata, one for each side. Unfortunately, design and implementation of communication protocols are difficult tasks: race and error conditions are especially hard to handle.

  Although one should expect that this complexity can be avoided with RMI, this is only partially true. In RMI, there is no way to access instance variables of remote objects. If an algorithm requires access to a variable of a remote object, the RMI programmer has to add at least one access function for that variable. For arrays a collection of access functions is needed (for the array, for individual dimensions, for single array elements). Moreover, RMI cannot be used for static methods and static variables. These restrictions often require significant changes to given code, before RMI can be used.

  Other code changes are caused by additional RMI requirements:

  - For every remote object that is to be used, an interface has to be declared that lists the available methods that can be called from remote.
  - Both in the new interface and in the class implementation, all these methods are required to throw a `RemoteException`. Therefore, all places in the code where remote methods are called have to be changed manually since the new exception must be handled.

    For example, let `foo` be a method of an RMI object that can be called from remote. Then the programmer must surround every call of `foo` by a `try` statement that catches the `RemoteException`.

    ```
    try {
      server.foo(...);
    } catch (RemoteException e) {
      ... what to do?
    }
    ```

    Whereas remote exceptions are a useful approach for wide area client-server applications, they do not occur in closely connected cluster settings. Whereas the extra code for registering and accessing the server can be hidden in a layer of abstraction, it is not easy to do so for ubiquitous method calls.

    Another disadvantage is that standard interfaces of the JDK, e.g., `java.lang.Enumeration` can no longer be implemented by remote methods.
  - In general, remote classes extend one of RMI's classes, e.g., `UnicastRemoteObject`. Since Java does not provide multiple inheritance, this may require a complete re-organization of the relationship between classes. (Instead of extending RMI classes, the programmer can explicitly add code that otherwise is inherited. However, this approach exposes even more internal details of the RMI mechanisms.)
  - Additionally, all JVMs must install the RMISecurityManager to use RMI. This is not only another task for the programmer but it prevents the use of other security managers, the user might prefer.

  Section 4 presents in detail how JavaParty achieves transparent method invocation.

- **Flexibility and Locality.** JavaParty specifically addresses the locality problem: If objects reside locally, they are used locally. Instead of expensive OS overhead a much cheaper single pointer indirection is sufficient. Since JavaParty objects can migrate, locality and thus performance can be improved.

  Quantitative measurements show that locality must be exploited: The invocation of an empty method via a pointer indirection takes about 0.7 μs for a Java 1.1.1 interpreter on a Sun Ultra which is about 18% higher than without indirection. A comparable RMI call of an empty method of a server that resides on the same physical machine takes 2.8 ms and is thus about 4000 times slower.

  If the hardware topology is variable there may be situations where some clients and the server reside in the same JVM on the same physical machine. Neither the socket versions nor the RMI version can take advantage of unexpected locality. Even worse, neither approach can exploit locality as a means of optimization at all. Since neither approach is intended for automatic object placement there will be no optimization for taking advantage of locality in future.

For the socket versions, communication is handled by the OS even if client and server reside on the same host. There is no way to avoid that messages travel through the protocol stack.

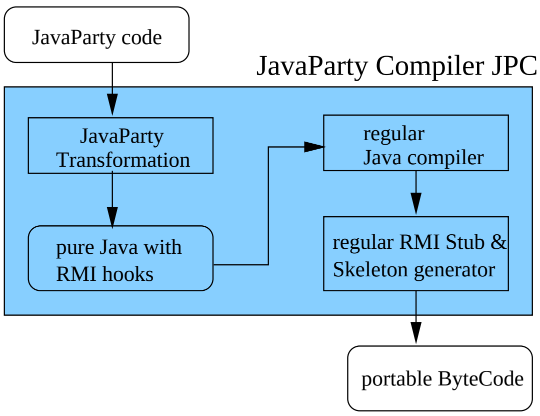
RMI is implemented on top of the socket layer and thus inherits the same disadvantages. But since stub and skeleton objects are used to implement RMI's remote method invocation, RMI could potentially do better. Instead of the stub-socket-socket-skeleton overhead to access a target object, the real target object could be addressed directly if it is local. Unfortunately, RMI does not do this. Moreover, there are no mechanisms in RMI to migrate objects and thus to increase locality on purpose.

Section 4 presents in detail how JavaParty implements migration and exploits locality.

4 Design and Implementation

We have implemented JavaParty as a pre-processing phase into our Java compilers EspressoGrinder [17] and Pizza [18]. Both Java compilers are freely available and have thousands of non-commercial and commercial users world wide. Alternatively, the JavaParty transformation can be used stand-alone to generate regular Java code that is then fed through any Java compiler, e.g., `javac`.

JavaParty currently uses RMI as a target and thus inherits some of RMI's advantages, e.g. distributed garbage collection. As shown in the diagram below, JavaParty code is transformed into regular Java code plus RMI hooks. The resulting RMI portions are then fed into Sun's RMI compiler (stub & skeleton generator).

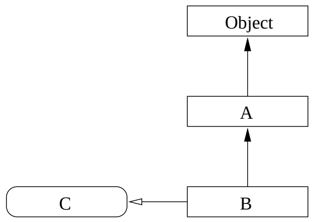


In the following sections we present the central ideas used for the transformation of JavaParty into Java plus RMI hooks. Many details of the transformation (e.g., object equality, `this`, synchronization, etc.) must be omitted.

The transformation is presented by means of the following example.

```
remote class B extends A implements C {
  T x = I;                  // instance variable
  static U y = J;           // static variable
  T foo(V z) { P }          // method
  static void foo2() { Q }  // static method
  static { R }              // static block
  B(T z) { S }              // constructor
}
```

The original class hierarchy is shown below. Solid arrows indicate subclassing by means of `extend`, open arrows represent the implementation of interfaces with `implements`. Boxes are used for classes, ovals are used for interfaces.



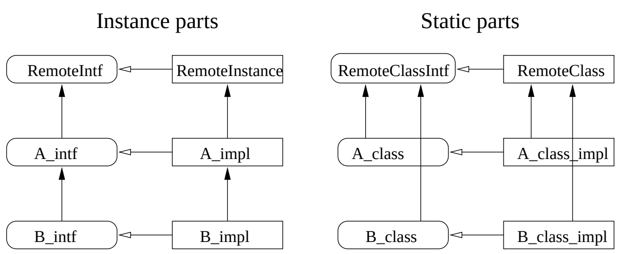
Instance Part and Static Part

The implementation of a class can be separated into those methods and variables that are static, i.e., that are the same for all objects of that class, and instance methods and variables that may have different values for each object. The semantics of the static class part are challenging to implemented in a distributed setting.

Whereas static parts cannot be accessed through RMI, they can be used in JavaParty. To achieve this, a remote class `B` is implemented by two RMI classes `B_impl` for the instance parts and `B_class_impl` for the static parts. Both classes are RMI classes. The situation is shown in the diagram below.

We mentioned above that RMI requires the declaration of an interface for each RMI class. This interface must declare all methods that can be called remotely. Therefore, in addition to the implementation classes `*_impl` there are two interfaces `B_intf` and `B_class`.

The following diagram presents a still incomplete version of the hierarchy of the resulting classes. The missing classes are called "handles" and will be discussed below. The top layer has classes that are provided by JavaParty's runtime system. These classes extend or implement RMI classes as appropriate.



For the static parts of class `B` the following code is generated:

```
interface B_class extends RemoteClassIntf {...}

class B_class_impl extends RemoteClass
implements B_class {
  U y;
  // Initialization of the class object
  protected void _init() {
    y = J;       // static variables
    toRemote(R); // static block
  }
  public void foo2() throws RemoteException {
    toRemote(Q)
  }
  // constructor for instance part
  public B_intf _new(T z) throws RemoteException {
    return new B_impl(z);
  }
}
```

In the generated code, RemoteExceptions are thrown as required by RMI. For each class that is loaded dynamically, a single object of B_class_impl is created on one of the hosts. (The host is determined by JavaParty's distribution strategy.) This single object implements the static variables and the static methods. The _init() method is called by the runtime system for initialization. In this and in the following code fragments we use toRemote to indicate that the code transformation is not presented in detail.

The last routine _new(T z) must be explained in some detail. JavaParty requires that objects can be created on remote hosts. There are two ways to implement that. One approach is to create the object locally and then migrate it to its final destination.

The cheaper and more elegant way, though, creates an object right at its intended destination. Without changing the Java virtual machine the only way of doing this is to call a regular class constructor at the destination. And this is what the _new routine does: It calls the constructor B_impl and returns a remote reference to the new object. Therefore, an instance of B_class_impl must be present on all hosts that need to create objects of type B_impl. However, only one of these instances actually implements the static parts.

The instance part of class B leads to the following two classes. (The implementation of B_impl is incomplete for ease of explanation; it will be completed below.)

```
interface B_intf extends A_intf {...}

class B_impl extends A_impl implements B_intf {
  T v = I; // instance variable
  public T foo(V z) // instance method
  throws RemoteException {
    toRemote(P)
  }
  // constructor
  B_impl(T z) throws RemoteException {
    toRemote(S)
  }
  public final T _get_B_v() //access method
  throws RemoteException {
    return v;
  }
  public final T _set_B_v(T _x) //access method
  throws RemoteException {
    return v = _x;
  }
  // in case T is a numeric base type:
  // access method
  public final T _inc_B_v(T _x, boolean _postfix)
  throws RemoteException {
    T _e = v;
    v += _x;
    if (_postfix) return _e; else return v;
  }
}
```

In the above code there is an instance variable v, a method foo, and a constructor B_impl. Three additional methods implement access functions, their names incorporate package, class, and variable names to deal with Java's different name resolution strategies for variables and methods. As required by RMI, all remote methods throw a RemoteException. If T is a numeric base type, then an additional access routine is created to efficiently implement ++, --, +=, and -=. In the example, we do not show special access routines for arrays (whole, individual dimensions, elements), for this, and others.

Handles and Locality

The careful reader might have noticed that the transformed code as presented so far no longer contains the original class B. Since existing code might be using B, this would result in inconsistencies. Moreover, remote methods throw RemoteExceptions that are still unhandled.

This problem is solved by a handle object B that hides the four RMI B_* classes/interfaces from users:

```
class B extends A implements C {
   ...
  T foo(V z) {          // instance method
    while (true)
      try { return ((B_intf)ref).foo(z); }
      catch (MovedException _e)
          {_adaptRef(_e);}
      catch (RemoteException _e)
          {_handleRemoteException("B.foo",_e);}
  }
  static void foo2() {  // static method
    try { ((B_class)RuntimeEnvironment.
          getClassObj("B")).foo2(); }
    catch (RemoteException _e)
          {_handleRemoteException("B.foo2", _e);}
  }
```

```
  ...
}
```

The class hierarchy of the handles is identical to the original class hierarchy shown above. For all instance and static methods of the original class there are methods with the same signature that do not throw any new exceptions. An incoming call is passed on to the appropriate remote object. For that purpose, handle classes internally hold a reference `ref` to access the instance part. The runtime environment is used to reference the static part of the class.

In addition to passing on method calls, handle objects deal with `RemoteExceptions`. Moreover, handles are used to hide migration, see below.

If the object `B_impl` does not reside on a remote host but is located locally, the handle sets the reference `ref` to directly point to the local object. Thus, no RMI overhead is needed at all and locality can be exploited at the cost of a single pointer indirection.

Object Migration

If a remote object that implements an instance part is moving to a different host, a proxy is left behind. If a method call arrives at the proxy, a `MovedException` is thrown back to the caller. Together with the exception, the caller is informed about the new location of the moved object. The caller uses this information to update the internal reference `ref` and to call the method at the new location. This explains the `while` loop in method `foo` of the above code fragment.

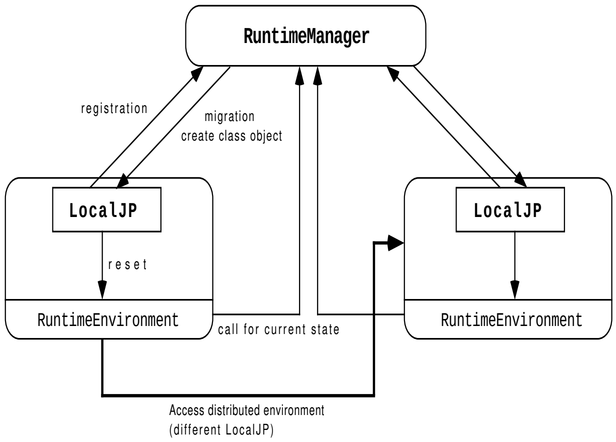
Since we have to make sure that objects are not moved while their methods are executed, the transformation of `B_impl` must be refined as shown below: the original method body is included in a `try` statement and surrounded by `_enter()` and `_leave()`. We only complete `foo(V z)`, the other methods require identical completions.

```
class B_impl extends A_impl implements B_intf {
  ...
  public T foo(V z)
  throws MovedException,
  throws RemoteException {
    _enter();
    try { toRemote(P) }
    finally { _leave(); }
  }
  ...
}
```

For actually migrating an object, the runtime system offers two options. An object can move to the position of the caller or it can be moved to the location of a different remote object. A migration is possible if no method is executing on the object, no other migration is in progress, and the object does not have its `resident` flag set. Then, the object's internal state is serialized into a byte array which is sent to the runtime system on the target host. The receiving runtime system unpacks the byte array and returns an RMI stub reference to the runtime system of the original host. Upon receipt, the original host completes the migration; for future method invocations the new reference will be thrown.

Runtime System

The runtime system consists of a central component, called *RuntimeManager*. In addition, each host runs a *LocalJP* that is registered at the manager. A host and its LocalJP can be added dynamically to the system. The manager knows all LocalJPs and knows the location of all class objects, i.e., for each class that is loaded the manager knows which host implements the static part. This information is replicated in the LocalJPs to reduce manager load. Neither manager nor LocalJPs need to know the location of individual remote objects. LocalJPs are needed to call constructors in class objects and to implement either side of a migration.



Current Shortcomings

Since JavaParty is implemented on top of RMI and since there are both remote classes and non-remote classes, some inconsistencies can occur:

- In method invocations, RMI passes remote objects by reference. Non-remote objects however are copied. Therefore, if a non-remote object is passed as parameter that itself refers to a structure of non-remote objects, then the whole graph of objects is copied to the recipient. If the receiving method changes the graph, then there will be different versions of it in the net.

  However, this is not as bad as it seems: Only objects can be copied that implement the interface `java.io.Serializable`. For method invocations, the compiler can often issue warnings if non-remote serializable objects may be passed to remote objects. If the programmer intends to change the copied data he either has to accept inconsistencies or must change the passed objects to be remote themselves.

- RMI requires that all methods must be public so that they can be called remotely. Thus, we have to weaken some of the access privileges for variables and methods.

  Since we cannot solve this problem outside of RMI, we do our best to keep it as small as possible. During compilation the JavaParty compiler checks that no access rights are broken; as usual, any access violations will cause error messages. Only after semantic analysis, the methods in the generated classes are made public. Therefore, a fair programmer does not suffer from the different access rights. However, a potential intruder from within the firewall who has access to the cluster may be able to find a way to call methods of existing objects, that where intended to be private.

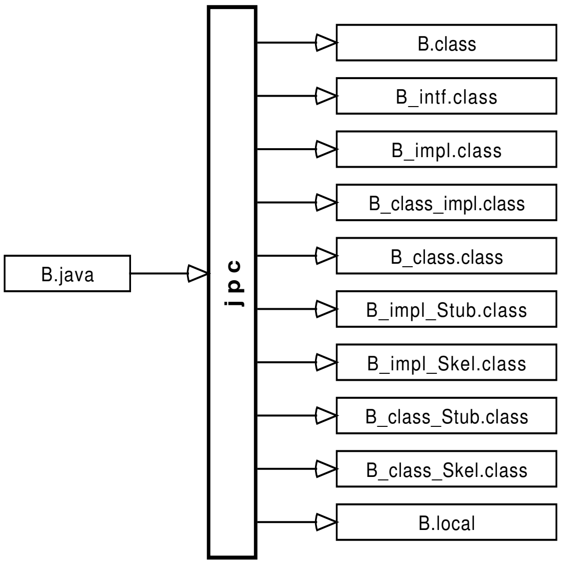
  We consider this problem to be acceptable for our prototypical implementation. A potential future implementation of JavaParty that is not based on RMI may be able to adequately guard private fields.

- If the signature of a class is changed, all its subclasses must also be recompiled since the RMI stubs and skeletons otherwise remain in an incompatible state.

- From a single JavaParty class `B`, ten Java Bytecode files are generated. We have already mentioned `B`, `B_intf`, `B_impl`, `B_class`, and `B_class_impl` above.

  Since `B_impl` and `B_class` are RMI files, the RMI compiler generates stubs and skeletons (`B_impl_Stub`, `B_impl_Skel`, `B_class_Stub`, and `B_class_Skel`).

  Finally, for separate compilation and the interaction of remote classes with non-remote classes during semantic analysis, we need an additional helper class `B.local` that is only used during compilation.



5 Related Work

- **Concurrent object-oriented languages.** From over a hundred existing imperative concurrent object-oriented languages (COOL) surveyed in [19] more than half do not consider problems of distribution and locality at all. The reasons are different: Some languages have only been implemented in a prototypical way on a single workstation, where network latencies do not occur; their developers have mainly been interested in the design of coordination mechanisms and a proof of concept. Other languages are restricted to shared memory multiprocessors, they rely on the cache systems provided by those machines.

  Most of the other languages are used to do research in concurrency coordination constructs. Threads and explicit synchronization as used in Java and JavaParty are not optimal for object-oriented languages because this approach suffers from various types of inheritance anomaly [15].[5]

  JavaParty has several advantages that most of the well known COOLs do not have: Since JavaParty is almost identical to Java, it can immediately be used by an army of Java programmers whereas other COOLs often are difficult to learn. Other advantages are inherited from Java as well: Portability across and instant availability on almost all hardware platforms, future performance improvements due to intensive work on all areas of the Java environment (byte code optimization, just-in-time and native compilation, garbage collection, etc.)

- **Parallelism in Java.** Although thread based parallelism is available in Java some researchers consider it inappropriate and lacking in expressive power. Some research groups therefore added data parallelism to Java [9] others added multiple paradigms [11, 13]. Some of these systems require additional machine dependent libraries or non-portable Java virtual machines. In contrast to those, JavaParty remains as close to Java as possible and runs in any standard Java environment.

  JavaParty currently does not offer any means for data parallelism. However, work is nearing completion that fills this gap: We are currently adding a `forall` statement that is pre-processed into threaded execution with remote threads by means of another source-to-source transformation.

- **Object Migration.** The positive effects of object migration have for example been studied in the Emerald project [10]. The JavaParty group currently stud-

[5] A very basic introduction to inheritance anomaly: The problem is that the lines of code that implement the synchronization requirements may be spread across all methods of a class. If a subclass has slightly different synchronization needs, inheritance anomaly is likely to occur: then instead of inheriting methods from the parent, nearly all methods must be recoded in the subclass. However, in the re-implementations, the algorithms themselves remain unchanged, just the synchronization code lines are modified. Code duplication results in higher maintenance efforts.

ies the integration thereof into compile-time and runtime optimization. None of the systems mentioned above offers object migration.

- **Target platform.** Currently, the implementation of JavaParty is based on RMI which is part of the standard JDK distribution. Therefore, JavaParty programs run on all major platforms, including Solaris, Windows, and NT.

  There are several alternatives: CORBA offers multilingual elements and is therefore not closely coupled to Java. Therefore, use of CORBA would require us to implement a lot of functionality that is already provided with RMI, e.g., a distributed garbage collector.

  "Horb" [8] is similar to RMI and offers a CORBA-like distributed environment. Hence, the results of the comparison of JavaParty versus RMI apply to Horb as well. On the other hand, Horb – claiming to have less runtime overhead than RMI – could have been used as a target for the implementation of JavaParty. But since RMI is part of the JDK 1.1 distribution and there are chances that RMI will eventually outperform Horb, JavaParty's usability is better if based on RMI.

  We refrained from implementing our own basic communication platform because we want JavaParty to instantly run on all major platforms.

- **Other remote Java objects.** We know of two other systems that try to implement transparent remote objects in Java.

  In contrast to JavaParty, "Remote Objects in Java" (ROJ) [16] introduces a new keyword `remotenew` that must be used to create objects on a specific remote host. The programmer is in charge of object placement. Since objects cannot migrate there is no way to enhance and exploit locality. The new keyword is mapped to a new Bytecode opcode. This opcode is implemented by an extended Java virtual machine and requires a specific interpreter. Therefore, ROJ cannot take advantage of progress in just-in-time compilation.

  ROJ method arguments are restricted to base types, i.e., it is not possible to pass object references. This restriction would have made it much more complicated to port our benchmark programs than it has been even with the socket and RMI versions.

  An interesting idea is that ROJ does not rely on a common file system. Whereas JavaParty uses the standard network to access Bytecode files, ROJ ships Bytecode. By shipping Bytecode, a single resulting Bytecode file is sufficient instead of ten.

  "Java/DSM" [22] is an implementation of Java on top of the Treadmarks distributed shared memory system [12]. Java/DSM requires special implementations of the Java virtual machine since objects must be allocated on the heap. In addition, Java/DSM has to struggle with heterogeneous environments which are already solved by our RMI approach. Moreover, since Java/DSM relies on Treadmarks for efficient caching and locality, we expect to get better performance since our approach which is based on compile time analysis can achieve more informed decisions.

6 Conclusion

JavaParty is an elegant way to program clusters of workstations and workstation-based parallel computers with Java. JavaParty programs are significantly shorter than equivalent programs based either on explicit socket communication or on RMI, they adapt more flexibly to varying network configurations and can exploit locality. JavaParty's runtime performance is comparable to RMI's performance. It will be much better once optimizations have been implemented.

We have presented transformation templates used by the JavaParty pre-processor to explain how various goals are met in practice.

JavaParty, consisting of a pre-processor generating regular Java, a complete compiler generating Bytecode, a runtime system, and some utilities, is freely available upon request for non-commercial projects. For more details and downloading information see `http://wwwipd.ira.uka.de/JavaParty`.

Acknowledgements

We would like to thank other members of the JavaParty group for their support of the JavaParty environment, especially Matthias Jacob who implemented the Veltran operator.

References

[1] Thomas E. Anderson, David E. Culler, and David A. Patterson. A Case for NOW (Network of Workstations). *IEEE Micro*, 15(1):54–64, February 1995.

[2] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jarov N. Seizovic, and Wen-King Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29–36, February 1995.

[3] J. Clearbout and B. Biondi. Geophysics in object-oriented numerics (GOON): Informal conference. In *Stanford Exploration Project Report No. 93*. October 1996. http://sepwww.stanford.edu/sep.

[4] John T. Feo, editor. *A Comparative Study of Parallel Programming Languages: The Salishan Problems*. Elsevier Science Publishers, Holland, 1992.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Mass., 1994.

[6] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, Reading, Mass., 1996.

[7] James Gosling, Frank Yellin, and The Java Team. *The Java Application Programming Interface*, volume 1 – Core Packages. Addison-Wesley, Reading, Mass., 1996.

[8] Satoshi Hirano. Horb – the magic carpet for network computing. http://ring.etl.go.jp/openlab/horb/, 1996.

[9] Susan Flynn Hummel, Ton Ngo, and Harini Srinivasan. SPMD programming in Java. *Concurrency: Practice and Experience*, June 1997.

[10] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[11] L. V. Kalé, Milind Bhandarkar, and Terry Wilmarth. Design and implementation of parallel Java with global object space. In *Proc. of Conf. on Distributed Processing Technology and Applications*, Las Vegas, Nevada, 1997.

[12] P. Keleher, A. L. Cox, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proc. 1994 Winter Usenix Conf.*, pages 115–131, January 1994.

[13] Pascale Launay and Jean-Louis Pazat. Integration of control and data parallelism in an object oriented language. In *Proc. of 6th Workshop on Compilers for Parallel Computers (CPC'96)*, Aachen, Germany, December 11–13, 1996.

[14] Doug Lea. *Concurrent Programming in Java – Design Principles and Patterns.* Addison-Wesley, Reading, Mass., 1996.

[15] Satoshi Matsuoka and Akinori Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 107–150. MIT Press Cambridge, Massachusetts, London, England, 1993.

[16] Nataraj Nagaratnam and Arvind Srinivasan. Remote objects in Java. In *IASTED Intl. Conf. on Networks*, January 1996.

[17] Martin Odersky and Michael Philippsen. Espresso Grinder. http://wwwipd.ira.uka.de/∼espresso, 1996.

[18] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Proc. 24th ACM Symposium on Principles of Programming Languages*, January 1997.

[19] Michael Philippsen. Imperative concurrent object-oriented languages. Technical Report TR-95-050, International Computer Science Institute, Berkeley, August 1995.

[20] Sun Microsystems Inc., Mountain View, CA. *Java Remote Method Invocation Specification, beta draft*, 1996.

[21] Thomas M. Warschko, Joachim M. Blum, and Walter F. Tichy. The ParaStation Project: Using Workstations as Building Blocks for Parallel Computing. In *Intl. Conf. on Parallel and Distributed Processing, Techniques and Applications (PDPTA'96)*, pages 375–386, Sunnyvale, CA, August 9–11, 1996.

[22] Weimin Yu and Alan Cox. Java/DSM: A platform for heterogeneous computing. *This issue*, 1997.