OpusJava
A Java Framework for Distributed High Performance Computing*†

Erwin Laure

Institute for Software Science
University of Vienna
Liechtensteinstrasse 22, A-1090 Vienna, Austria
E-Mail: erwin@par.univie.ac.at

Abstract

The past few years have dramatically changed the view of high performance applications and computing. While traditionally such applications have been targeted towards dedicated parallel machines, we see the emerging trend of building "meta-applications" composed of several modules that exploit heterogeneous platforms and employ hybrid forms of parallelism. In particular, Java has been recognized as modern programming language for heterogeneous distributed computing. In this paper we present OpusJava, a Java based framework for Distributed High Performance Computing (DHPC) that provides a high level component infrastructure and facilitates a seamless integration of high performance Opus (i.e., HPF) modules into larger distributed environments. OpusJava offers a comprehensive programming model that supports the exploitation of hybrid parallelism and provides high level coordination means.

1 Introduction

Recent trends in hardware, in particular in interconnection technologies, have paved the way to the exploitation of heterogeneous, distributed computing platforms for advanced scientific applications. While traditionally high performance applications have been targeted towards dedicated parallel machines, we see the emerging trend of building "meta-applications" [29] that are composed of several modules and exploit heterogeneous platforms. The efficient exploitation of parallelism within and across modules is of paramount importance for such applications that include problem solving environments (PSEs) [4, 5], where algorithmic modules are integrated into distributed scientific computations, collaborative systems, and distributed supercomputing applications (also referred to as "Grand Challenges") [10].

In this paper we propose a Java based framework that provides high level support for *Distributed High Performance Computing (DHPC)*. DHPC applications are often built by composing existing software components that exploit specialized computing and algorithmic resources [10, 11]. A typical example of a DHPC scenario is a parallel simulation that harnesses (clusters of) parallel machines for its computation, receives the simulation data from a distributed database, and visualizes the results online on specialized graphical workstations.

Common characteristics of DHPC applications include:

*The work described in this paper was partially supported by the Special Research Program SFB F011 "AURORA" of the Austrian Science Fund.

†Accepted for publication in *Future Generation Computer Systems*, Elsevier, 2001.

1. multiple modules need to cooperate in solving a problem,
2. parallelism can be exploited within and across modules,
3. modules are written in different languages, and
4. modules are dynamically instantiated on different, heterogeneous platforms.

Currently, there is a lack of high level support for DHPC and designers of DHPC applications are facing a variety of low level systems and services that often do not integrate well. For instance, Java and Corba are two well known systems for distributed computing, however, they do not sufficiently support high performance parallel computing. In particular, it is a non-trivial task to integrate parallel applications, written e.g. in MPI or HPF, efficiently into distributed Java or Corba applications.

In this paper we present OpusJava, a Java [12] interface to the coordination language Opus [6, 7] that provides a high level, comprehensive model for DHPC. OpusJava combines the strengths of Opus (in particular the integration of HPF [16] style data parallelism with flexible task parallel features) with the strengths of Java (in particular the dynamic networking features). OpusJava provides a high level component model that hides low level details such as communication, data format conversion, or thread synchronization.

While Opus provides efficient support for coupling modules which exploit hybrid forms of parallelism on homogeneous systems (properties (1) and (2) from above), it lacks support for language interoperability and heterogeneous platforms (properties (3) and (4)). This is mainly due to the Fortran/HPF centric approach [23]. OpusJava is designed to bridge the gap between efficient high performance computing and dynamic distributed computing. In particular, OpusJava provides:

- a high level Java component model for distributed computing, and
- language interoperability with HPF by supporting a seamless integration of Opus modules.

Java was chosen as base language because of its excellent support for heterogeneous platforms. It provides platform independence, a common data representation, communication via RMI or sockets, multithreading, dynamic class loading, object serialization, and reflection, each of which is indispensable for our system. Although other communication mechanisms such as Corba could have been employed as well, we designed our first prototype exclusively based upon Java to allow a rapid prototyping of the concepts introduced.

Java is, moreover, maturing as programming language and is increasingly employed for high performance applications [20]. OpusJava supports both, the development of new high performance Java modules, by providing a high level distributed computing framework, and interoperability with legacy (in most cases Fortran) code through its interface to Opus.

The OpusJava framework is pure Java, without modifications to the language or the JVM. Thus, any existing Java code, even compiled byte-code, can be re-used within an OpusJava application and such applications will run on any standard Java system. The interface to Opus requires some modifications to the Opus compilation and runtime support, though. This issue will be discussed in detail in Section 6.2.2.

The remainder of this paper is organized as follows: We illustrate the salient features of OpusJava with an abstract case study in Section 2 and 5. After a short review of the Opus Language in Section 3 we present OpusJava in Section 4. The implementation design for both, Opus and OpusJava as well as their interfaces, are discussed in Section 6. In this section we also report on preliminary experiences with our prototype implementation. We contrast OpusJava to related approaches in Section 7 and end the paper with summarizing comments.

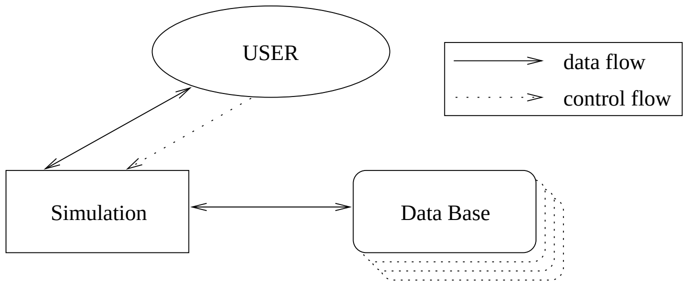

Figure 1: Structure of a typical DHPC application

2 Case Study Part 1: Online Monitoring and Computational Steering

We illustrate our OpusJava design with a prototypic DHPC application. Rather than discussing a specific problem, we present the design of a generic scenario that focuses on the coordination aspects among components. Thus, we can omit specific algorithmic details and illustrate the high level OpusJava programming model in an easily comprehensible way.

Figure 1 illustrates the structural design of a typical DHPC application: the user monitors and steers a numerical simulation which receives its data from a distributed database.

It is desirable that the user monitors and steers the application remotely, perhaps using a Java applet. The simulation itself needs to harness massively parallel machines; hence, it is most likely implemented in MPI, HPF, or Opus. The distributed database is most likely implemented in C or C++ and provides an SQL-like query language.

Such DHPC scenarios are nowadays typically realized by hard-wiring the components and providing problem specific interfaces which are tailored to the application and can hardly be reused. OpusJava, instead, provides a high level component model that gives the programmer an easy means for expressing component interaction, hybrid forms of parallelism, and synchronization at a high level.

An OpusJava implementation of this scenario is presented in Section 5, after an introduction to the salient features of Opus and OpusJava in the next two sections.

3 Overview of the Opus Language

Opus [6, 7] introduces a small set of extensions to HPF [16] designed to coordinate an efficient parallel execution of multiple, independent HPF program modules. At the heart of these extensions is an abstract data type called a *ShareD Abstraction*, or *SDA*, whose purpose is to provide a means for encapsulation of data and methods (procedures) which act on this data. Opus borrows heavily from Fortran 90 in the definition and use of SDAs by extending the Fortran 90 module concept and from monitors in shared memory languages in that an active method has exclusive access to the data of an SDA. SDAs may exploit data parallelism in that the internal data of SDAs as well as the data of SDA methods may be distributed.

Viewed as an abstract data type, an *SDA type* specifies an object structure, containing data and the methods which manipulate this data. By *creating* an instance of an SDA type an *SDA object* (or simply SDA) is generated. At creation of an SDA, the resources on which it will execute are allocated, and all SDA data is allocated and initialized in order to establish a well-defined initial state. During its *lifetime* an SDA executes asynchronously in a separate address space and can be accessed through calls to its methods.

An SDA method can be invoked *synchronously*, where the caller is blocked until control returns, or *asynchronously*, where the caller does not have to wait for the completion of the method. Explicit synchronization is possible via *event* variables bound to asynchronous method invocations. More specifically, an SDA may *test* (or poll) an event in a non-blocking fashion, thus getting information about a method's status, or it may *wait* until a method has finished its execution. No two methods of an SDA are allowed to execute in parallel; thus each method has *exclusive access* to the data of the SDA. A method may have an associated *condition clause*, specifying a logical expression which guards the method's activation.

HPF syntax is used to support data parallelism within an SDA. Thus, each SDA type definition may contain an optional processors statement, as defined for HPF procedures [16]. This allows the internal data structures of the SDA to be distributed across the processors using the distribution and alignment directives of HPF. The dummy arguments of the SDA methods can also be distributed using the rules applicable to procedure arguments in HPF.

The execution of an Opus program can be thought of as a system of SDAs in which an SDA executes a method in response to a request from another SDA.

4 OpusJava

OpusJava is a Java framework that provides Opus functionalities. Thus, it can be seen as the Java binding of Opus. Similar to Opus, OpusJava provides a high level component model (SDAs) with plug compatibility among components, a hybrid parallelism model, and high level coordination means.

Moreover, OpusJava facilitates the integration of HPF modules into distributed Java applications by providing an interface to Opus. Language interoperability is furthermore ensured by the standardized Java Native Interface (JNI). OpusJava does not pose any restrictions to the Java programming style. In particular, Java's multithreading features may well be exploited within SDAs. However, exploiting multithreading requires the user to ensure a proper program behavior by appropriate thread synchronization. Moreover, OpusJava properties, such as the monitor semantics of SDAs, may be circumvented by multithreading techniques which may result in an undefined program behavior.

The OpusJava framework, whose API is modeled after the Opus syntax to provide a common look-and-feel, provides both, access to Opus SDAs, and Java SDAs with their salient features. In particular, OpusJava allows any Java object to be created as an SDA. The OpusJava framework hides all low level details such as socket communication, RMI, JNI, and synchronization; generic interfaces are provided, thus not even interface specifications via IDLs are necessary.

By exploiting advanced Java features such as the reflection mechanism, we were able to avoid language extensions and IDLs in our design of OpusJava. This fact favors a wide spread use of OpusJava by making the framework easily available on any Java machine. Moreover, the absence of compilation requirements allows the integration of dynamically loaded byte-code modules into OpusJava applications.

OpusJava introduces only two new classes visible to the user: the final class `SDA`, which is a stub for a remote SDA object, and the final class `Event`, which is used to reference asynchronous method invocations. The detailed API specification can be found in the appendix.

A remote SDA is created by instantiating a new object of type `SDA`. Its constructor has 4 arguments: the name of the remote SDA, an array of input parameters for the remote constructor, the machine name on which it should be allocated, and the number of processors to be used. Resources of a multi-processor machine are currently allocated in a system dependent manner. We plan to provide the user with high level means for guiding the resource allocation, similar to the Opus resource allocation mechanisms [6]. In particular, means for *indirect* resource allocation (i.e. relative to existing SDAs) will be developed. After its creation, a remote SDA is accessible via the `SDA` stub.

The `SDA` stub provides methods for calling remote methods in a synchronous (`call`) and asynchronous (`spawn`) way. The arguments of these methods are the remote method name and an array of input parameters for the remote method. A `terminate` method for gracefully destroying remote objects is provided as well. Although Java does not provide means for destroying objects

```
// SDA creation
SDA foo = new SDA(< class name >,
                  < input arguments >,< machine >,
                  < number of processors >);

// synchronous method invocation
Object result = foo.call(< method name >,
                         < input arguments >);

// asynchronous method invocation
Event e = foo.spawn(< method name >,
                    < input arguments >);

// explicit synchronization
e.waitEvent();

// use of results
Object result = e.getResults();
```

Figure 2: SDA creation and method invocation

and leaves all cleaning mechanisms to the garbage collector, SDAs may be native (HPF) objects and thus support for explicit termination is desirable.

The `call` method blocks the caller until the results are available, while the `spawn` method returns an `Event` object immediately and allows the caller to resume its execution. An event is a pointer to an asynchronous remote method invocation and can be used for synchronization later on. Event objects provide two synchronization methods: `testEvent` and `waitEvent`. The former one immediately returns the current status of the method the event is bound with (i.e. *finished* or *not finished*), while the latter one blocks the caller until the referenced method has finished its execution and the results are available.

The results of a method invocation are returned as a Java object. While in the case of synchronous calls the regular Java result mechanism can be used, asynchronous calls need a slightly different treatment. The execution of the `spawn` method results in the immediate return of an event object. This object is not only used as a synchronization means, but also as a container for outstanding results. In particular, an event object has a private field of type `Object` which is used for storing the method results. The public method `getResults` may be used for obtaining the results from the event. This method is blocking, i.e. it waits until the asynchronous method execution has finished and all of its results are available. Hence, any invocation of `getResults` implicitly synchronizes the caller with the asynchronous method invocation.

Events not only provide *data based* synchronization (i.e. implicit synchronization when results of an asynchronous request are being accessed), but also *control based* synchronization that is independent of any data access (by *testing* or *waiting* on an event). In that respect, events are comparable to Multilisp's *Futures* [14], but differ from *Futures* in similar Java systems [2, 19] that only provide data based synchronization.

Figure 2 illustrates the creation of an SDA object, synchronous and asynchronous method invocations, as well as explicit synchronization and access to results of asynchronous calls.

Similar to Opus, OpusJava provides twofold synchronization mechanisms: first, asynchronously spawned methods are bound to `Events` with which SDAs can explicitly or implicitly synchronize as described above; and second, OpusJava supports the Opus concept of conditional method execution. This feature provides a high level means for ensuring correct program behavior by guarding method activations with a logical condition. While Opus introduces a new keyword for *condition clauses* (`WHEN`), OpusJava utilizes a specific naming convention. A condition clause guarding a Java method must be a logical function, without parameters, whose name is the name of the guarded method postfixed with "`_cond`".

```
public class Buffer {

   private int data[];        // the buffer
   private int size = 0;      // the current size
   private int max;           // the maximum size


   // constructor
   public Buffer(int s) {
      max = s;
      data = new int[max];
   }

   public void put(int i) {
      data[size++] = i;
   }

   public int get() {
      return data[--size];
   }

   // condition clause for put
   public boolean put_cond() {
      return (size < max);
   }

   // condition clause for get
   public boolean get_cond() {
      return (size > 0);
   }
}
```

Figure 3: Bounded Buffer with Condition Clauses

A typical example for the use of condition clauses is a bounded buffer. Condition clauses provide a high level mechanism to guarantee a consistent state of the buffer, i.e. to avoid over- and underflows. Figure 3 illustrates such an example. With the help of the two condition clauses *put_cond* and *get_cond*, which implement over- and underflow checks, respectively, the buffer is guaranteed to be consistent: *put* and *get* methods may be invoked in any order and any number of times—the OpusJava framework takes care of activating them only after the associated condition clause has been satisfied (cf. Section 6.2).

OpusJava supports the dynamic class loading features of Java. Hence, the bytecode of SDAs may be dynamically loaded from user specified resources on demand.

For implementing OpusJava, advanced Java features such as multithreading, object serialization, reflection, and communication over RMI and sockets are exploited. The JNI is used for interfacing Opus. We discuss the implementation design in detail in Section 6, after illustrating OpusJava with the continuation of our example of Section 2.

5 Case Study Part 2: An Implementation with OpusJava

Recall the scenario given in Section 2: a user monitors and steers a simulation running on a parallel machine that receives its data from a distributed database. This scenario asks for three obvious SDAs: the *user interface* (probably the main program), the *simulation*, and the *database*. In order to decouple the simulation from the user interface to a greater extent, we introduce another SDA, a *buffer*, which is used for storing the current status of the simulation and the simulation parameters, as well.

```
SDA TYPE simulator(buffer,database)

  SDA (buffer_type) buffer
  SDA (database_type) database
   ! simulation data
   ! HPF PROCESSOR directives
   ! HPF DISTRIBUTE directives

CONTAINS

  SUBROUTINE simulate

    DO WHILE (.not. converged)
       params = buffer%get_parameters()
       ...
       buffer%set_status(...)
    END DO

  END SUBROUTINE simulate

END SDA TYPE
```

Figure 4: The HPF Simulator

We assume that the high performance component of our scenario, the *simulation*, is realized with Opus in order to exploit the available resources best. However, the interface to the *database* is probably easily facilitated with the help of a Java wrapper, since Java provides a standardized database interface (JDBC). The *user interface* that will probably be operated over the Internet is as well implemented in Java. The *buffer* could be realized either in Opus or OpusJava. Since the interaction *simulator/buffer* is more time critical than the interaction *buffer/user interface*, and since method invocation among two Opus SDAs is less time consuming than method invocation among Opus and OpusJava (cf. Section 6.3), we decided to implement both, the *simulator* and the *buffer* in Opus. Note that the *simulator* might well exploit hybrid parallelism by being composed of multiple data parallel SDAs.

Figure 4 shows the declaration of the simulator while Figure 5 depicts the declaration of the buffer. The simulator has access to two SDAs, the buffer and the database, and operates a simulation loop that repeatedly retrieves parameters from the buffer, executes one iteration of the simulation, and dumps the current status to the buffer. For sake of brevity we omit the interaction with the database, which is realized in a similar manner as the buffer interaction. The simulation proceeds until the problem converges. The buffer, on the other hand, is used to store the status data as well as the parameters and offers 4 methods for accessing its data: `get_parameters` and `set_parameters` for reading/writing the parameters and `get_status` and `set_status` for reading/writing the status data, respectively.

Figure 6 sketches the Java code for monitoring and steering. We show only the code necessary for interacting with the simulation and omit all the user interface related parts. The application starts with the creation of the SDAs: the buffer (line 4), the database wrapper (5), and the simulator (7). The simulator receives a pointer to the buffer and the database as parameters. Note that in the constructor parameters of the simulator, the number of nodes available to the HPF-simulation is contained as well. Both, the simulator and the buffer, are allocated on the same machine, while the database wrapper resides on the physical location of the database. Subsequently, the simulation parameters are initialized (9) and the simulation is spawned asynchronously (11). The asynchronous method invocation is associated with an event that is used in the monitoring loop (13) for testing whether the simulation has finished. Within that loop, the status is retrieved from the buffer (14), visualized, and, dependent if a tuning of the parameters is necessary, the parameters in the buffer are updated (17).

```
SDA TYPE buffer_type
 ! status data
 ! parameters

CONTAINS

  FUNCTION get_parameters
    ...
  END FUNCTION

  SUBROUTINE set_parameters(...)
    ...
  END SUBROUTINE

  FUNCTION get_status
    ...
  END FUNCTION

  SUBROUTINE set_status(...)
    ...
  END SUBROUTINE

END SDA TYPE
```

Figure 5: The Buffer

6 Implementing Opus and OpusJava

In this section we discuss the implementation design of both, Opus and OpusJava, and present their interface. We first summarize an ongoing effort that is expected to lead to an efficient realization of Opus and discuss then our design considerations for OpusJava. In particular, the Opus implementation is targeted towards architectures that provide a parallel virtual machine view (e.g. through MPI) while OpusJava is designed to support the coordination of components which are distributed over heterogeneous platforms. We present both, the Java framework as well as its interface to Opus and report on first experiences we gained with a prototype implementation at the end of the section.

6.1 Implementation Design of Opus

Recently, an implementation effort for Opus was initiated [22]. One of the main design goals, apart from having an efficient and portable implementation, was the integration with the *Vienna Fortran Compiler (VFC)* [1] so that we do not have to re-implement any of the HPF compiler work. Moreover, we re-use low-level runtime components like thread and communication modules.

The transformation of Opus source code into an executable program is a multi-stage process. The Opus compiler transforms the code into blocks of HPF code with calls to the Opus Runtime System (*ORS*). The resulting code is further processed by VFC which produces Fortran 90 code with calls to the VFC runtime system. This code is eventually compiled and linked with both the VFC runtime system and the ORS.

The ORS [22] provides runtime representations for SDAs and method invocation requests (MIs) as well as higher abstractions of low level system components (such as pthreads [17] and MPI [24]). Moreover, methods for buffering MIs on the callee side, which is due to the conditional execution of methods, are provided. MI-buffers are realized using two FIFO queues called *MI-queues*: the *inqueue* buffers incoming MIs while the *outqueue* keeps track of issued MIs.

The Opus compiler transforms the code of SDAs into separate threads of control to facilitate an efficient execution of multiple SDAs mapped onto overlapping resources. However, within an SDA

```
public class Monitor {

  public static void main(String[] args) {
    // create SDAs
    SDA buffer = new SDA("buffer_type",machine,1);
    SDA database = new SDA("database_type",database_machine,1);
    Object[] args = new Object[] {buffer,database};
    SDA simulator = new SDA("simulator", args, machine, nr_nodes);

    // initialize parameters
    buffer.call("set_parameters", params);

    // spawn the simulation
    Event ev = simulator.spawn("simulate");

    // monitor
    while (! ev.testEvent()) {
      status = buffer.call("get_status");

      // visualize and analyze the status

      if (update_necessary)
        buffer.call("set_parameters",params);
    }
  }
}
```

Figure 6: Main Program

thread there is need for multiple threads as well because a component that is able to receive MIs independently of ongoing method executions is required. Hence, one can think of a hierarchical thread structure where an SDA thread is subdivided into two independent threads communicating via a shared memory segment. As shown in Figure 7 the components of an SDA are:

- A shared memory area, in which the *MI-queues* are allocated.
- The *Server Thread* which receives MIs, constructs execution records (the runtime representation of MIs) and places them into the inqueue.
- The *Execution Thread* which retrieves records from the inqueue, evaluates the condition-clauses, and executes the respective methods which are compiled to internal subroutines of the execution thread.

Both threads execute asynchronously parallel; the only synchronization required is a mutually exclusive access to the shared queues. The structure of these threads is fixed in so-called *compilation templates* which are filled in by the compiler (see [22] for a detailed discussion).

A prototype of the Opus Runtime System and the Opus compiler, which focuses on the tasking features of Opus, is available on clusters of SUN-SPARC workstations, on the QSW(Meiko)-CS2, and on the Cray T3E. Preliminary tests show encouraging runtime results which can be found in [22, 23].

6.2 Implementation Design of OpusJava

The OpusJava framework comprises two main components: the Java framework that facilitates Java SDAs and the interface to Opus. While the former one is designed as a pure Java package, the latter one consists of a Java part, JNI interfaces, and Fortran components. All the parts of the Opus interface are generated automatically by the Opus compiler.

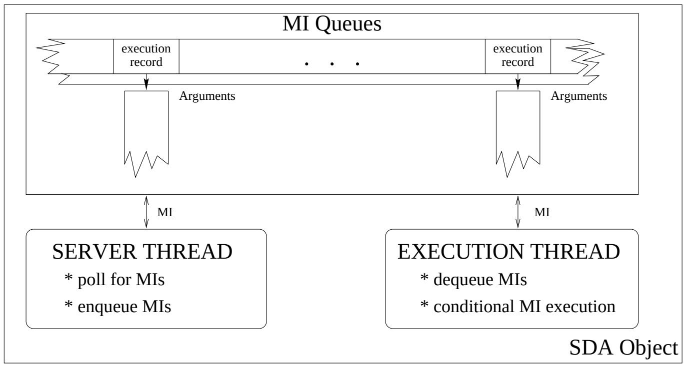

Figure 7: Structure of an SDA Object

6.2.1 The Java Framework

The OpusJava framework consists of three main components:

- SDA Implementation Skeletons, which are generic skeletons for SDA objects,
- SDA Stubs, which are pointers to remote SDA objects, and
- SDA Servers, which are daemons, responsible for creating and terminating SDA objects.

SDA Implementation Skeleton Every SDA object is wrapped by a skeleton that handles all its interactions with other components. In particular, methods of the SDA are invoked via the skeleton rather than directly in the SDA. Skeletons are implemented through the final Java class `SDAImpl`. In order to allow remote access to SDAs, skeletons are registered in the RMIRegistry. The most important method exported is `insert`, which is responsible for receiving and temporarily storing method invocation requests.

`SDAImpl` skeletons are realized in a manner similar to Opus SDA objects (cf. Section 3 and [22]). In particular, a skeleton contains an *inqueue* and an *outqueue* for managing *execution records* and operates an *execution thread* which is spawned upon creation. Method invocation requests are processed in a similar way as in the Opus *Server Thread*: the `insert` method creates an *execution record* which contains references to the method to be executed, the input parameters, and some internal management data. Java's reflection mechanism is exploited for the creation of execution records. In particular, the correctness of the supplied method signature is validated. The execution record is then inserted into the *inqueue* from where it is retrieved by the *execution thread*. After ensuring the validity of the request by evaluating the associated *condition clause*, the execution thread invokes the method in the SDA object. The skeleton takes also care of transferring results of a method execution back to the caller via plain socket communication.

The SDA Stub A remote SDA object is accessible via a local SDA stub that is realized by the final class `SDA`, introduced in Section 4. These stubs are in fact pointers to SDA objects to which they contain remote references. SDA stubs take care for transmitting requests as well as receiving results synchronously or asynchronously. The constructor of a stub issues an RMI call to the *SDA Server* (cf. next paragraph) on the indicated machine and thus initiates the construction of a new remote SDA object together with an associated `SDAImpl` skeleton which is registered in the RMIRegistry. A permanent RMI connection for subsequent method invocations is built between the local `SDA` stub and the remote `SDAImpl` skeleton.

Activations of the `call` and `spawn` method result in an RMI request to the skeleton of the callee. At the same time, the stub opens a socket for receiving the results and spawns a thread that listens on this socket. Thus, the caller is able to resume its execution immediately in case of an asynchronous request while a synchronous request blocks the caller until the results are available. An asynchronous request returns an `Event` object, that is linked to the remote execution record of the request and thus allows the retrieval of information about the status of the request.

An activation of the `terminate` method allows SDAs to terminate gracefully. In particular, the termination of an SDA requires its synchronization with all outstanding method invocations. Subsequently, the `terminate` method of the SDA Server is activated which results in the deletion of the SDA from the RMIRegistry.

SDA Server The SDA Server is a daemon that has to be installed on any system participating in an OpusJava program execution. This daemon is responsible for creating and terminating SDA objects for which two RMI methods (`create` and `terminate`) are exported.

The `create` method of the SDA Server, which is activated by the constructor of an `SDA` stub, has 3 arguments: the name of the SDA type, an array of input parameters for the SDA's constructor, and the number of processors to be used. Any Java object can be created as an SDA and Java's reflection mechanism is exploited to check whether appropriate classes are available. If not, dynamic class loading can be used to load the classes from user specified resources on demand. Subsequently, the new SDA object is created using the supplied arguments for its constructor. An associated skeleton is created and registered in the RMIRegistry as well. The registered name is sent back to the caller, thus completing the `create` method.

The `terminate` method simply deletes an SDA skeleton from the RMIRegistry as sketched above. Thus, Java's garbage collector may claim the associated memory when necessary.

Figure 8 summarizes and illustrates the mechanisms involved in the creation of SDAs and in method invocations:

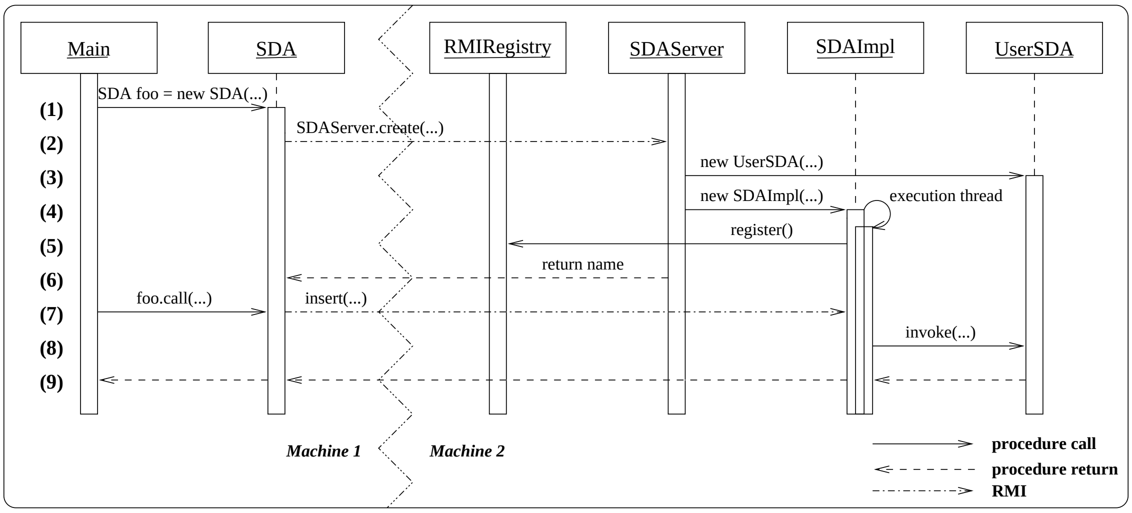

Figure 8: UML Sequence Diagram for SDA Creation and Method Invocation

1. A new `SDA` object is created on the caller's side.
2. The SDA Server of the target machine is contacted via RMI.
3. This results in the creation of the user SDA object.
4. It also results in the creation of an associated SDAImpl object, which contains a reference to the user SDA object; the execution thread is spawned.
5. The SDAImpl object is registered in the RMIRegistry.

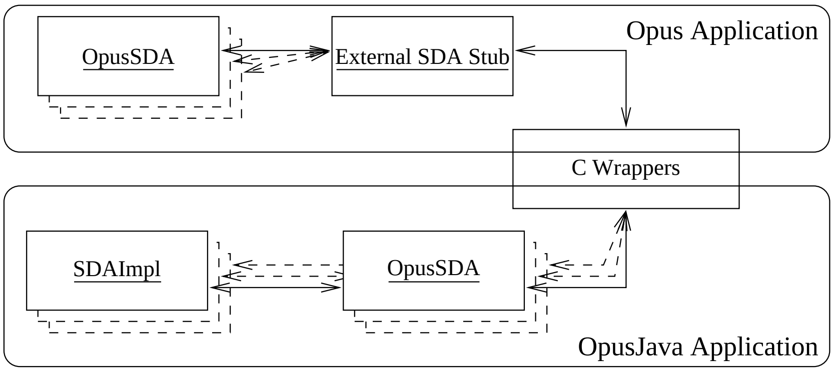

Figure 9: Interplay of Opus and OpusJava

6. The registered name is sent back to the caller.
7. For invoking a method, the SDA stub issues an RMI call to the `insert` method of the associated SDAImpl skeleton.
8. The *execution thread* of the skeleton executes the method.
9. It sends back the results, thus completing the method invocation.

Events Apart from the above described main components, the OpusJava framework implements `Event` objects that are pointers to asynchronous method invocations. Events have a `result` field which is used to store the output parameters of the associated method and can be accessed through the `getResults` method. The activation of the `testEvent` method results in an inquiry of the status of the associated method. `true` is returned, if the method has already finished its execution, otherwise the result is `false`. An activation of the `waitEvent` method blocks the caller until the remote method has finished its execution and its results are available. `getResults` implicitly activates `waitEvent` in order to guarantee a well defined state of the `result` field which is returned to the caller.

6.2.2 Interfacing Opus with OpusJava

In order to make Opus (i.e. Fortran 90 or HPF) SDAs accessible from OpusJava, the Opus compiler generates stubs and wrappers that route requests from an Opus module to an OpusJava module, and vice versa, and take care for the needed data conversion and marshaling. While the OpusJava framework provides generic interfaces and postpones correctness checks to runtime exploiting Java's reflection mechanism, the Opus compiler requires information about the interfaces of all the SDAs (hence also Java SDAs) in a program at compile time. For this purpose, *SDA interfaces* designed along the lines of Fortran 90 procedure interfaces [18] are provided. With help of these interfaces the Opus compiler is able to generate all necessary components that allow an Opus SDA to interact with OpusJava SDAs.

In particular, three modules are generated by the Opus compiler:

- A Java counterpart for every Opus SDA type,
- C wrappers for interfacing Fortran with Java, and
- an Opus SDA stub that routes requests from/to Opus to/from the outside.

Figure 9 illustrates how these components, whose functionality is described below in more detail, work together in an OpusJava application.

Java SDA Classes For every Opus SDA type, a Java counterpart is generated by the Opus compiler. This counterpart is a Java class with the same name and methods as the Opus SDA. A new object of this type is created as an SDA (thus together with an associated skeleton) for every Opus SDA object. Within an OpusJava application, methods of these SDAs can be invoked just as of any other OpusJava SDA, however, such requests are transmitted to the respective Opus SDA via C wrappers rather than being served directly. Hence, these Java objects act as OpusJava wrappers for Opus SDAs.

C Wrappers OpusJava defines a generic JNI interface for method invocation requests (thus all the JNI compilation can be done when installing the framework; no additional compilation is needed for particular applications). The Opus compiler generates C wrappers employing this interface which build the link between OpusJava and Opus applications. In particular, the wrappers marshal and unmarshal method arguments and construct the required Opus execution records. The OpusJava `SDA` interface requires method arguments to be passed as an array of `Objects`. This array is transferred to the C wrappers, where it is split up into its components in a way that is conforming to the Opus method signature. In addition, multi-dimensional arrays are packed into appropriate Fortran 90 arrays. The results of a method invocation are marshaled in the same way: multi-dimensional arrays are transformed to Java equivalents and all results are packed into an `Object` (which might be an array in case of multiple output parameters, that are possible in Fortran by declaring arguments with `INTENT OUT` or `INOUT`). This object is eventually returned as result to the Java SDA. Note that the interface only allows Fortran arrays and basic types as parameters. Advanced Fortran features such as derived types are not supported, however, appropriate interfaces are being developed in the ongoing Fortran 200x standardization process and will be employed within our framework once they are available.

External SDA Stub An Opus application accessible from OpusJava provides a stub through which external requests are routed. This stub acts as a multiplexer that transfers OpusJava requests obtained from the C wrappers directly to the respective Opus SDA and vice versa. For an Opus application, the stub appears as a normal SDA, thus any request admissible to SDAs can be posed to the stub, as well. But rather than serving the request directly, the stub routes it through the C wrappers to the associated OpusJava SDA for service. Note that while every SDA object has its Java counterpart in order to be accessible, only one stub is required for an Opus application. The stub is able to serve all the SDAs of an Opus application.

6.3 Experiments

We implemented a prototype of the OpusJava framework using the Java 2 platform and extended our Opus compiler [22] to provide the necessary interfaces. With this prototype environment we realized the most significant parts of the monitoring application discussed above and measured the time needed for synchronous and asynchronous method invocation between (a) 2 OpusJava SDAs and (b) an OpusJava SDA and an Opus SDA, respectively. We measured round-trip times between two SUN Ultra 10 workstations, running JVM 1.2.1, connected by fast-ethernet. Thereby, we used different input/output data sets ranging from 0 to 2000 floats[1]. In the synchronous case we measured the time the caller is blocked in the call, while in the asynchronous case the time from initiating the call until the return of an immediately followed `waitEvent` on the associated event was taken. Figure 10 shows the mean time elapsed and, for comparison reasons, the pure MPI based Opus results as well. Note, that these timings reflect the full OpusJava mechanisms as described above rather than pure communication times.

While the performance of the OpusJava implementation for synchronous calls without parameters comes close to the MPI based Opus performance, there is a drastic decrease in performance when more data is being passed. This overhead is mainly due to the well known object serialization overhead of Java. Fortunately, there are already attempts to reduce this overhead [27] and

[1] We give the array size rather than bytes because of Java's RMI and object serialization mechanisms employed.

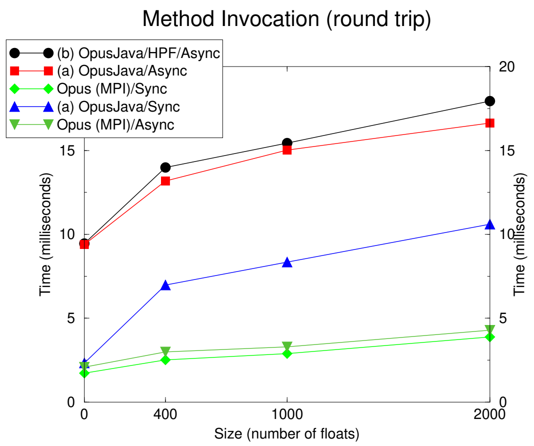

Figure 10: Elapsed Time for Method Invocation

we will study these efforts for our future developments. There is also some improvement potential for Java's RMI mechanism [25] although the performance problems are mainly not due to RMI itself but to the involved object serialization.

This object serialization overhead is also visible in the timings for asynchronous method invocations. However, asynchronous method invocation suffers from an additional performance penalty of a factor of approximately 4 with respect to Opus. This additional overhead is mainly due to the less efficient thread handling and context switching in the JVM employed. We expect this problem to disappear with future, optimized JVM releases.

The uppermost curve shows the elapsed time for asynchronous method invocation between an OpusJava and an Opus (i.e. HPF) SDA. The overhead for crossing the language boundaries is almost negligible and hence, the overall performance of the OpusJava framework is mainly bound to the quality of the underlying JVM.

7 Related Work

A number of systems have been developed to support distributed high performance computing. They range from low level services and middle-ware to high level languages. The community has also seen the benefits of Java for coordinating distributed tasks resulting in a large and rapidly growing number of frameworks. However, most of these Java systems provide advanced Java solutions, but do not tackle language and legacy system interoperability. A good overview of systems supporting distributed and meta-computing is provided by the Datorr Project [8].

In this section we briefly review a representative selection of these approaches and highlight the differences and benefits of the OpusJava model.

Globus [9] The Globus system is a metacomputing toolkit which enables computing in distributed heterogeneous environments. Globus provides a set of low level services from which application developers can select to meet their needs. This set of services currently consists of 7 components for *resource management*, *communication*, *information*, *security*, *health and status*, *remote data access*, and *executable management*.

While these services provide some useful functionality for designers of DHPC applications, Globus does not offer a comprehensive programming model supporting the specific properties of such applications. Moreover, parts of Globus (such as Nexus) often prevent the exploitation of libraries tuned to specific machines (like native MPI implementations). However, some components of Globus, such as the resource manager or the security module, may well be employed within OpusJava.

Corba [26] The *Common Object Request Broker Architecture (Corba)* was defined recently to support portable and interoperable applications. The central part of Corba, the *Object Request Broker (ORB)* acts as a mediator between clients asking for a service provided by a server. All interaction is handled by the ORB; thus clients and servers are not directly connected and do not need to know details about their counterpart such as the physical location or the implementation language. Using a Corba specific *Interface Definition Language (IDL)* a clean interface can be provided for all required services.

Although Corba provides a clean architecture for coupling different modules and for exploiting heterogeneous systems it has some weaknesses when taking the requirements of DHPC applications into account. In particular, Corba does not provide explicit support for data parallelism and does not provide adequate support for high level coordination of tasks. Nevertheless, Corba may well be used as an underlying service for OpusJava (e.g. as replacement of the employed RMI mechanism).

Legion [13] Legion is a software architecture that supports a single, coherent virtual machine on top of the heterogeneous structure exhibited by the Internet. The key concept of Legion is a distributed object model. Objects can be distributed over the available computational resources in a user guided way; object interaction takes place via method invocation. Legion also provides support for parallelism in that either parallel languages like MPL, BFS, or Java are supported directly, or that other parallel modules (such as HPF or MPI codes) can be wrapped with an object and used within Legion. Similar to Corba, Legion supports language interoperability by using IDLs.

Legion has similar aims as OpusJava, however, it lacks high level coordination of asynchronous remote procedure calls. In addition, Legion has problems with the integration of optimized parallel code. In particular, Legion requires parallel libraries like MPI to be emulated using the underlying Legion runtime library. Hence, vendor optimized implementations cannot be exploited. Moreover, parallel objects appear as simple sequential ones to other Legion objects which prevents e.g. optimized communication schedules between all components of such objects. Opus already supports parallel objects and a port of this feature to OpusJava is planned in the near future.

WebFlow [15] WebFlow is a three-tier system that comprises a graphical frontend, a Corba based middle tier and the Globus toolkit as backend. Therefore, WebFlow can be regarded as a high level, visual user interface to Globus. Initially, WebFlow was designed with a Java (RMI) based middle tier. This design was changed to Corba because of the more enhanced language interoperability features of Corba.

Although WebFlow provides a high level graphical programming and program composition tool for DHPC, its design based on Globus suffers from the very same problems in the context of DHPC. Moreover, WebFlow does not provide a comprehensive programming model supporting hybrid forms of parallelism. In particular, data parallel objects are, similar to Legion, encapsulated within a single WebFlow module.

JavaParty [28] JavaParty introduces transparent remote objects to Java by declaring them with the class modifier `remote`. Hence, JavaParty can be seen as a significant simplification of Java's RMI mechanism. Moreover, JavaParty may outperform pure Java RMI applications by exploiting locality among objects. JavaParty is locality transparent: the mapping of objects on the available platforms in a network is performed by the compiler and runtime system, though explicit mappings can be enforced by the programmer.

JavaParty provides an easy to use and efficient distributed component model for Java. It lacks, however, interoperability support with other high performance languages and systems. If language interoperability is required, the user has to tackle with JNI and with the problem of interfacing high performance systems, such as HPF or MPI, explicitly. Moreover, JavaParty currently leads to code explosion: for a single remote JavaParty class, ten Java bytecode files are generated. OpusJava, instead, only requires a thin *SDAImpl* layer for any OpusJava object.

Ajents [19] Ajents is a Java system that supports parallel, distributed and mobile applications. Ajents allows the creation of remote objects and supports synchronous and asynchronous RMI operations on them by providing generic user-level functions. As within OpusJava, RMI functionality is transparent. Furthermore, Ajents has some basic support for checkpointing and object migration.

Although the API of Ajents is close to OpusJava, Ajents does not provide the flexible synchronization mechanisms available in OpusJava. In particular, synchronization with asynchronous RMIs is only possible by accessing the result object (which is encapsulated within *Futures*). An access to such a Future will block the caller until the results are available. OpusJava, instead, provides synchronous as well as asynchronous means for accessing the execution status of asynchronous method invocations via the *Event* object. Moreover, OpusJava provides *conditional method execution* which is missing in Ajents.

Ajents' API for remote method invocations consists of a number of overloaded functions and restricts the number of parameters passed to a remote method. OpusJava does not introduce any such restrictions.

Finally, OpusJava provides language interoperability, in particular a clean interface to HPF, while Ajents does not provide any additional support beyond Java's JNI.

8 Conclusions

In this paper we have presented OpusJava, a comprehensive programming model that supports hybrid parallelism (task and data parallelism) in the context of a multi-lingual framework for dynamic distributed high performance computing. In particular, OpusJava provides high level communication and coordination means and allows a seamless and efficient integration of high performance Opus components into larger distributed Java based environments. These high performance components need not be changed and can still exploit highly optimized communication mechanisms available on their target platform. The user of OpusJava can focus on the algorithmic problem: all low level details such as communication, data format conversions, or JNI are completely transparent. Moreover, OpusJava employs Java's dynamic class loading features, which allow new modules to be plugged into a running application dynamically.

The OpusJava framework can furthermore be used as a high level component infrastructure for distributed Java programs. No modifications of the Java language or the JVM are necessary to run OpusJava. These benefits, however, have the drawback, that Java's reflection mechanism is exploited to check the validity of OpusJava requests at runtime. In particular, the generic OpusJava interfaces do not allow static type checking nor checking of method signatures within SDAs. These features could be integrated within OpusJava by means of an IDL, however at the expense of additional compilation effort.

Future work will focus on the optimization of OpusJava's method invocation implementation that is currently based upon RMI. In particular, optimizations which enhance Java's object serialization [27] and RMI [25] mechanisms will be employed. We will also investigate the use of Corba and JINI [21] in this context.

High level means for the specification of resource allocation strategies will be developed. These features will closely follow the existing Opus mechanisms [6].

Currently, OpusJava supports the creation of any Java object as an SDA. However, distributed Java objects are not yet supported. We will study the usefulness of the current development of HPJava [3] in this context.

In ongoing projects we will study the applicability of OpusJava to a number of application fields such as financial planning or distributed OLAP for high performance computing.

Acknowledgments

The author would like to thank Bernd Wender for many fruitful discussions on Java and the OpusJava design.

References

[1] S. Benkner. VFC: The Vienna Fortran Compiler. *Journal of Scientific Programming*, 7(1):67–81, December 1998.

[2] D. Caromel, W. Klauser, and J. Vayssiere. Towards seamless computing and metacomputing in Java. *Concurrency: Practice and Experience*, 10(11-13):1043–1061, 1998.

[3] B. Carpenter, G. Zhang, G. Fox, et al. Towards a Java Environment for SPMD Programming. In *Proceedings EuroPar '98*, LNCS, No. 1470, Southampton, 1998. Springer.

[4] H. Casanova and J. Dongarra. NetSolve: A Network Server for Solving Computational Science Problems. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(3):212–223, Fall 1997.

[5] H. Casanova and J. Dongarra. The use of Java in the NetSolve project. In *Proceedings of the 15th IMACS World Congress on Scientific Computation, Modelling and Applied Mathematics*, Berlin, 1997.

[6] B. Chapman, M. Haines, E. Laure, P. Mehrotra, J. Van Rosendale, and H. Zima. Opus 1.0 Reference Manual. Technical Report TR 97-13, Institute for Software Technology and Parallel Systems, University of Vienna, October 1997.

[7] B. Chapman, M. Haines, P. Mehrotra, J. Van Rosendale, and H. Zima. OPUS: A Coordination Language for Multidisciplinary Applications. *Scientific Programming*, 6(9):345–362, Winter 1997.

[8] Datorr. http://www.computingportals.org.

[9] I. Foster and C. Kesselman. The Globus Project: A Status Report. In *IPPS/SPDP '98 Heterogeneous Computing Workshop*, 1998.

[10] I. Foster and C. Kesselman, editors. *The Grid*. Morgan Kaufmann, 1999.

[11] D. Gannon et al. Developing Component Architectures for Distributed Scientific Problem Solving. *IEEE Computational Science & Engineering*, 5(2):50–63, April-June 1998.

[12] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesely, 1996.

[13] A. Grimshaw, W. Wulf, et al. The Legion Vision of a Worldwide Virtual Computer. *Communications of the ACM*, 40(1):39–45, January 1997.

[14] R. Halstead. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, 1985.

[15] T. Haupt, E. Akarsu, and G. Fox. WebFlow: A Framework for Web Based Metacomputing. In *Proceedings HPCN Europe 1999*, LNCS, No. 1593. Springer, 1999.

[16] High Performance Fortran Language Specification Version 2.0, January 1997.

[17] IEEE/ANSI. POSIX Threads Extensions. IEEE/ANSI 1003.1c-1995.

[18] ISO. Fortran 90 Standard. ISO/IEC 1539 :1991 (E).

[19] M. Izatt, P. Chan, and T. Brecht. Ajents: Towards an Environment for Parallel, Distributed and Mobile Java Applications. In *ACM 1999 Java Grande Conference*, San Francisco, California, June 1999.

[20] Java Grande Forum. http://www.javagrande.org.

[21] Jini Connection Technology. http://java.sun.com/jini/.

[22] E. Laure, M. Haines, P. Mehrotra, and H. Zima. On the Implementation of the Opus Coordination Language. *Concurrency: Practice and Experience*, 12(4):227–249, April 2000.

[23] E. Laure, P. Mehrotra, and H. Zima. Opus: Heterogeneous Computing With Data Parallel Tasks. *Parallel Processing Letters*, 9(2):275–289, June 1999.

[24] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 1.1*, June 1995.

[25] C. Nester, M. Philippsen, and B. Haumacher. A More Efficient RMI for Java. In *ACM 1999 Java Grande Conference*, San Francisco, June 1999.

[26] OMG. *CORBA/IIOP 2.2 Specification*, June 1998.

[27] M. Philippsen and B. Haumacher. More Efficient Object Serialization. In *Proc. International Workshop on Java for Parallel and Distributed Computing*, San Juan, Puerto Rico, April 1999.

[28] M. Philippsen and M. Zenger. JavaParty - Transparent Remote Objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, November 1997.

[29] L. Smarr and C.E. Catlett. Metacomputing. *Communications of the ACM*, 35(6):45–52, June 1992.

A The OpusJava API

In the following, the API for OpusJava, in particular for the classes `SDA` and `Event`, is specified. Note, that all the necessary exceptions are omitted for sake of conciseness. In presenting the API we follow closely Java's API style.

Class SDA

```
java.lang.Object
  |
  +--SDA
```

`public final class` **SDA**
`extends Object`
`implements Serializable`

> The SDA class is a stub for remote SDA objects. It establishes a permanent RMI connection to the remote SDA and provides methods for invoking methods of the remote SDA in a synchronous or asynchronous manner. Explicit support for terminating remote SDAs is provided too.

Constructors

`public` **SDA**`(String name)`

> Constructs a new SDA object and contacts the **`SDAServer`** of the local machine in order to create a new object of class **`name`** as an SDA.
> **Parameters:**
> **`name`** - the class name of the object to be created as SDA.

`public` **SDA**`(String name, Object[] args)`

> Constructs a new SDA object and contacts the **`SDAServer`** of the local machine in order to create a new object of class **`name`** as an SDA, passing **`args`** as input arguments for the remote constructor.
> **Parameters:**
> **`name`** - the class name of the object to be created as SDA.
> **`args`** - the input arguments for the constructor of class **`name`**.

`public` **SDA**`(String name, String machine, int processors)`

> Constructs a new SDA object and contacts the **`SDAServer`** of **`machine`** in order to create a new object of class **`name`** as an SDA, using **`processors`** processors
> **Parameters:**
> **`name`** - the class name of the object to be created as SDA.
> **`machine`** - the name of the machine on which the SDA should be created.
> **`processors`** - the number of processors to be used on **`machine`**.

`public SDA(String name, Object[] args,`
`String machine, int processors)`

Constructs a new SDA object and contacts the `SDAServer` of `machine` in order to create a new object of class `name` as an SDA, using `processors` processors and passing `args` as input arguments for the remote constructor.

Parameters:

`name` - the class name of the object to be created as SDA.

`args` - the input arguments for the constructor of class `name`.

`machine` - the name of the machine on which the SDA should be created.

`processors` - the number of processors to be used on `machine`.

Methods

`public Object call(String method)`

Invokes `method` in the remote SDA in a synchronous manner. Results are returned in `Object`.

Parameters:

`method` - the name of the remote method.

Returns:

The results encoded in an `Object`.

`public Object call(String method, Object args[])`

Invokes `method` in the remote SDA in a synchronous manner passing `args` as input arguments. Results are returned in `Object`.

Parameters:

`method` - the name of the remote method.

`args` - an array of input arguments.

Returns:

The results encoded in an `Object`.

`public Event spawn(String method)`

Invokes `method` in the remote SDA in an asynchronous manner. An `Event` is returned immediately, which acts as a reference to the call and contains also the results of the method.

Parameters:

`method` - the name of the remote method.

Returns:

A new `Event` object bound to the request.

`public Event` **`spawn`**`(String method, Object args[])`

Invokes **`method`** in the remote SDA in an asynchronous manner passing **`args`** as input arguments. An **`Event`** is returned immediately, which acts as a reference to the call and contains also the results of the method.

Parameters:

`method` - the name of the remote method.

`args` - an array of input arguments.

Returns:

A new **`Event`** object bound to the request.

`public void` **`terminate`**`()`

Terminates the remote SDA. In particular native SDAs are terminated by invoking their termination routines which free the memory in use. The RMI link is disconnected.

Class Event

```
java.lang.Object
  |
  +--Event
```

`public final class` **Event**
`extends Object`
`implements Serializable`

The Event class represents a link to asynchronously spawned methods of an SDA. It is typically only created as return value of **`spawn`** methods. Events allow queries about the status of an asynchronous method invocation and encode the results of such methods, as well. It provides methods for testing a method invocation's status in a non-blocking fashion, for waiting on the return, and for accessing the results.

Methods

`public boolean` **testEvent**`()`

Tests the status of a method invocation. If the method execution has already terminated, **`true`** is returned, otherwise the result is **`false`**.

Returns:

The execution status of a remote method invocation.

`public void` **waitEvent**`()`

Blocks the caller until the remote method referenced has finished its execution.

`public Object` **getResults**`()`

The results of an asynchronous method execution are returned. In the case the execution has not yet finished, the caller is blocked until the results are available.
