Chapter 4

XCAT based component architecture

4.1 The components choice

4.1.1 Components extending GrADSoft

GrADSoft is a modular software based on the GrADS design. To create the XCAT based component architecture for GrADSoft, it was natural to give a look at GrADSoft in the first place. As shown in Figure 2.1, the main GrADSoft modules are : Repository, Builder, Application Manager, Scheduler/Resource Selector, Dynamic Optimizer, Performance Monitoring Setup Module, Contract Monitor and Launcher.

So to choose the components that should be implemented, the functionality offered by each of these modules were carefully examinated. The Launcher is part of the Application Manager, so it is not a good component candidate. Likewise the Application Manager would not be an interesting component. We need to recall here than in the XCAT vision, an Application Manager is only a Jython script. That is why it will not be converted to a component. The Dynamic Optimizer and the Performance Monitoring Setup Module are not implemented by GrADSoft. The Contract Monitor is still an open research area. This makes them impossible to componentize.

So stays three candidates : Repository, Builder, Scheduler/Resource Selector. The three of them can be viewed as services. They can be running on different machines and wait for calls. So these three GrADSoft modules were chosen as components. The design and architecture of these XCAT based components will be detailed in Section 4.2.

4.1.2 Controller component

After these three components were chosen, the question of how they were going to interact together, what type of framework is needed to coordinate them and to make them communicate was raised.

The first thought was to use an Application Manager : a Jython script in the XCAT view. The Application Manager would launch the components and connects them together. But what is needed to do here is more sophisticated : the componentized software should be able to read the user's configuration file (an example

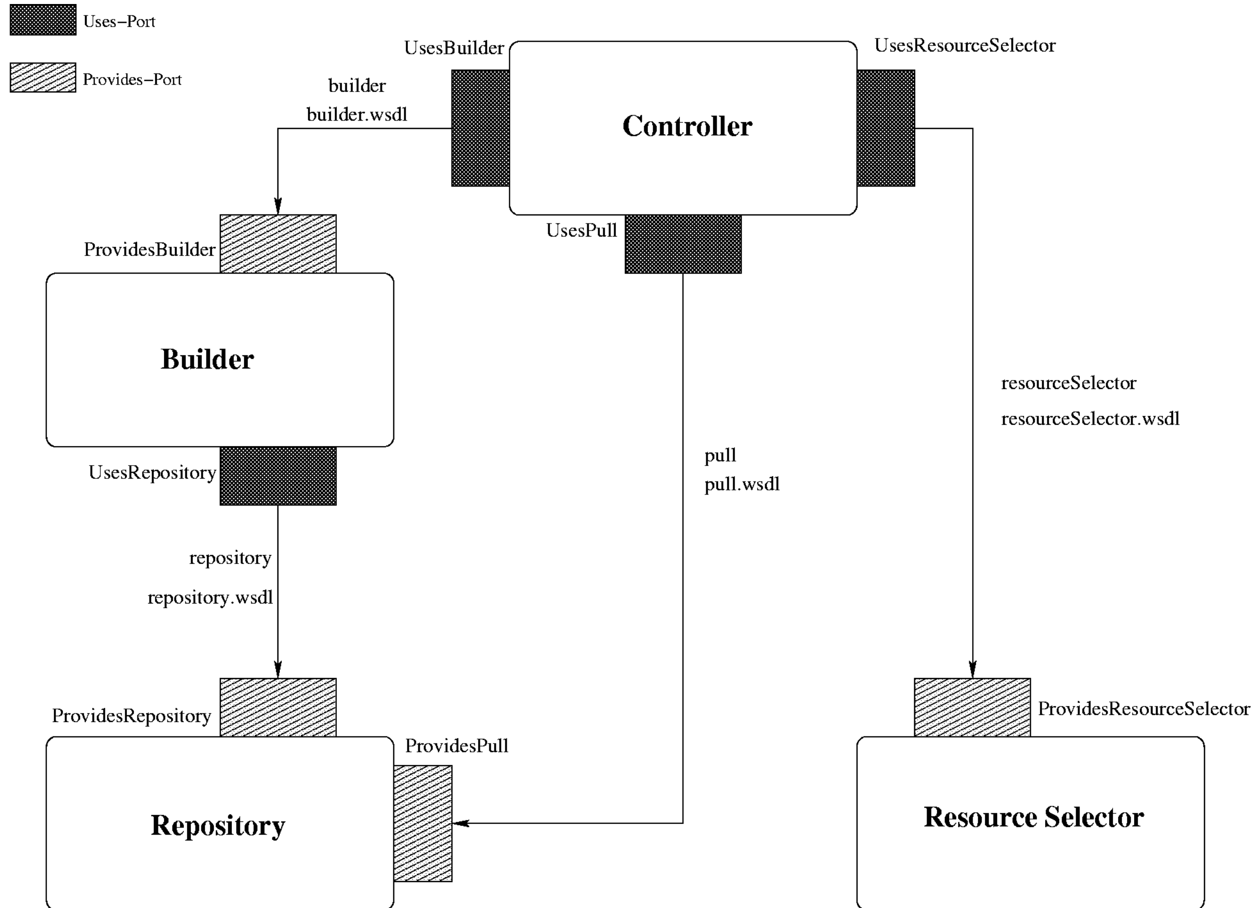

Figure 4.1: XCAT based component architecture choice

of such a file can be found in Figure 2.3), initialize the components that need an initialization, and call some provides-ports.

So it was decided that the Application Manager should do the simple components' instantiation and should establish the connections. The more complex tasks are then let to another component : the Controller. The Controller doesn't offer any provides-port, but it has as many uses-ports as the number of other components. These uses-ports will allow the Controller to : initialize the Repository, initialize the Builder, call the $GetAART()$ function provided by the Builder Component, and call the $GetBestVM()$ function provided by the Resource Selector Component. Figure 4.1 shows the chosen components and how they are linked together.

4.2 The components architecture

We used the following naming convention for naming ports. If a port interface is called Interface, then the provides-port corresponding to this interface is called ProvidesInterface and the uses-port is called UsesInterface (cf. Figure 4.1).

4.2.1 Repository Component

The Repository component offers the same functionalities as the GrADSoft object repository. The component Repository allows data to be stored and retrieved. The component has two provides-ports named : ProvidesRepository and ProvidesPull. Between the Repository and the Builder the interface is *repository*. Between the Repository and the Controller the interface is *pull*.

The port ProvidesRepository provides two methods : $putAART()$ and $getAART()$ which are used to store and retrieve AARTs. The Builder component can use these methods via UsesRepository to store and retrieve data.

The port ProvidesPull provides only one method : $init()$. The port UsesPull from the Controller component can invoke this method to initialize the Repository with a default or specific configuration.

4.2.2 Builder Component

The Builder component is designed to offer the same behavior as the GrADSoft object builder. Today, the Builder constructs only two predetermined AART objects (cf. section 2.2.2). This component has one provides-port called ProvidesBuilder and one uses-port called UsesRepository. Between the Builder and the Controller the interface is *builder*. Between the Builder and the Repository the interface is *repository*.

UsesRepository is connected to the provides-port (ProvidesRepository) of Repository by the interface *repository*. This port allows a remote method provided by the Repository to be invoked which calls $GetAART()$ to retrieve the AART. Today the $PutAART()$ method is never invoked by the Builder because the Repository does not currently provide storing capability.

ProvidesBuilder provides two methods : $GetAART()$ and $init()$. The $init()$ function is used by the Controller to initialize the Builder and the role of the $GetAART()$ function is to invoke the $GetAART()$ function provided by the Repository. It is done using the UsesRepository port.

4.2.3 Resource Selector Component

The Resource Selector component provides the same functionalities as the GrADSoft object resourceSelector. The Resource Selector provides functionalities to estimate the best set of Virtual Machines for a specific problem. This component has only one provides-port : ProvidesResourceSelector. Between the Resource Selector and Controller component the interface is *resourceSelector*.

The UsesResourceSelector port, which is a uses-port of the Controller, is connected to the ProvidesResourceSelector port. ProvidesResourceSelector provides two methods : $GetBestVM()$ and $GetBestVMs()$. The $GetBestVMs()$ function combines several types of data to estimate the best set of Virtual Machines. The $GetBestVM()$ function uses the $GetBestVMs()$ function to obtain a unique Virtual Machine. There is no $init()$ method here because the Resource Selector does not require any specific initialization. In other words, the constructor of Resource Selector is empty.

```
class Repository {
  public:
    /**
     * Returns an AART described by the 'aartReference' if it is found
     * in the repository, otherwise returns NULL.
     */
    AART * GetAART(const string & aartReference);
    /**
     * Adds an AART to the repository. Returns true if successful,
     * false if failure (which could be because this aart is already in
     * the repository).
     */
    bool PutAART(AART * aart);
  private:
    /**
     * List of AARTS contained in the repository
     */
    list <AART *> aartList;
};
```

Figure 4.2: GrADSoft Repository

```
package samples.idl.repository;
public interface Repository {
    public AARTObject getAART(String aartReference);
    public int putAART(AARTObject aart);
}
```

Figure 4.3: XCAT IDL Repository definition

4.2.4 Controller Component

The Controller component behaves as the GrADSoft application manager. This component can be viewed as a process that coordinates all the other components, so it has three uses-ports which are connected to the provides-port of the three other components described above. These three ports are : UsesBuilder, UsesPull, and UsesResourceSelector. The Controller does not have any provides-port.

UsesBuilder is connected to ProvidesBuilder by the interface *builder*. UsesPull is connected to ProvidesPull by the interface *pull*. And UsesResourceSelector is connected to ProvidesResourceSelector by the interface *resourceSelector*.

4.3 Code generation for each component

This section describes the three steps that have to be done to build components.

4.3.1 Java specification

This stage consists in writing the Java (the SIDL, cf. Section 3.3.2.3) specification of each port interface : builder, repository, pull, and resourceSelector. XCAT is aware of a few data types (int, float, string, dou-

```
enum TopologyType { NONE, SINGLE_NODE, VECTOR, MESH, NS_MESH, TREE, FCG };
/// A struct to store information about each dimension/level
typedef struct {
  list<Constraint *> Dim_Constraint_list;
  unsigned int Size_measure;
} Dimension;
class AART {
    private:
     /// One of NONE, MESH, NS_MESH, TREE, FCG, etc.
     TopologyType Topology;
     /// Number of dimensions or levels defined in AART
     unsigned int Dimensions_Levels;
     /// Constraints that apply to all processors in topology
     list<Constraint *> App_Constraint_list;
     /// Descriptors for each dimension/level in the AART
     vector<Dimension *> Dim_C_data;
     /// Name for the AART, for identification, storage, and retrieval
     string Reference;
}
```

Figure 4.4: GrADSoft AART

ble) and one data structure (array). Therefore modifications were made as follows to map non-recognized datatypes into XCAT-recognizable datatypes :

- boolean has been transformed into int
- list has been transformed into array
- vector has been transformed into array
- struct has been transformed into object
- enum has been transformed into int

We will now describe an example of such a mapping using SIDL. Figures 4.2 and 4.3 show respectively the GrADSoft specification for the repository object and its Java translation. Because the methods handle AARTs, we have to specify an AART object for Java which will be called AARTObject to avoid confusion. Figure 4.4 and Figure 4.5 show the translation between a GrADSoft AART object and an XCAT AART object (AARTObject). This work has to be done recursively for every new object declared into an AARTObject.

4.3.2 Code generation

Then the XML Schema Document Generator (cf. Section 3.1.2) takes the Java specification as input to perform the translation into the target component implementation language : C++. To do this, two types of XML files have to be written : mappingsInfo and metaInfo. MappingsInfo files bind the SIDL types used by an object declared in SIDL to the target language types. MetaInfo files specify which mappingsInfo files are needed to generate each of the output file. There are seventeen output files for the port interface and five files for each object, as shown in Appendix B. The user should not have anything to modify in the Pack/Unpack files. These files describe how objects should be packed to be sent on the wire and how they

```
package samples.idl.repository;
import java.io.Serializable;
public class AARTObject implements Serializable {
    private int Topology;
    private int Dimensions_Levels;
    private ConstraintObject[] App_Constraint_list;
    private DimensionObject[] Dim_C_data;
    private String Reference;
    public AARTObject() {}
}
```

Figure 4.5: XCAT IDL AARTObject definition

should be unpacked on the receiving side. However it is up to the user to implement the methods declared within the *object.h* file (where *object* is the AARTObject for example) into a corresponding *object.cpp* file.

4.3.3 Component implementation

We added five directories to the current GrADSoft distribution. The first four are for the components themselves : Builder, Repository, Controller, and ResourceSelector. The last directory (named *idl*) is for the port interfaces. Four subdirectories are made for the four interfaces previously described. Figure 4.6 shows us the directories organization. There is also a *doc* directory (not shown on the figure) containing this document.

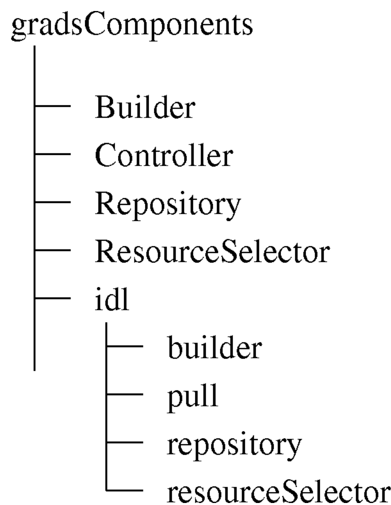

Figure 4.6: Directories list

4.3.3.1 Port interfaces

In each directory corresponding to an interface there are the seventeen generated files (cf. Section 3.3.3) plus the five files per object resulting from the transformation of GrADSoft objects into XCAT objects and the *object.cpp* file where the basic get*Attribute*()/set*Attribute*() functions are implemented.

4.3.3.2 Components

A few files have to be written to make the components useful and to generate the executable for each of them. Our component is here called *Component*.

*Component*Server.cpp is the file where the *main* function is located. All the ports are registered (at the same time an instance of each port is created), an instance of *Component* is declared and the *setServices*() function is called on it (cf. Section 3.3.5).

*Component*Component.h and *Component*Component.cpp contain the definition and the implementation of two methods : *setServices*() and *execute*(). The *execute*() function exists only if the component has a uses-port. It is used to invoke remote methods on other components.

For each provides-port, an *Interface*Impl.h file and an *Interface*Impl.cpp file have to be written. They contain respectively the definition and the implementation of the provided methods defined in the IDL.

4.4 Adapting an XCAT generated object to a GrADSoft usable object

XCAT-C++ has defined its own protocol of communication based on XSOAP : an implementation of the Java RMI API in C++ (cf. Section 3.2.1). This protocol imposes some restrictions concerning the types of element sent on the wire. More precisely complex data structures as array can not be sent without being wrapped. So an array declared in the SIDL is transformed during the code generation phase into another XSOAP specific type : SoapCppArray. SoapCppArray is defined as a container for arrays. It is the generic array for XSOAP which can be sent on the wire between two components.

The aim of a component can be reduced to this idea : invoke GrADSoft methods and send the invocation's result on the wire to another component. Communications between two components require specific XCAT objects as described in Section 4.3.1 and GrADSoft methods require GrADSoft objects. So a mechanism needs to be implemented to make XCAT objects and GrADSoft objects compatible. This mechanism is called **Adaptor** and it is used to transform GrADSoft objects into XCAT objects and vice versa : it **adapts** GrADSoft objects to XCAT objects and vice versa. For example a GrADSoft AART (defined Figure 4.4) has to be transformed into an XCAT AARTObject (defined Figure 4.7) to be sent to another component. An array of *object* needs to be transformed into a SoapCppArray containing the array's data, the number of elements and the information that the array is of type *object*.

4.5 The test cases

With this test case, we try to stay as close as we can to GrADSoft's test cases, which mainly consist of launching application managers. An example of a GrADSoft test case can be found in Appendix D. GrADSoft application manager does the following : it reads the user's configuration settings from the configuration file, calls the Builder to retrieve the AART corresponding to the application and calls the Resource Selector to retrieve an appropriate Virtual Machine. The XCAT component test case is in fact a real and complete execution of the whole process which goes from reading the user's configuration file to

```
class AARTObject : public xsoap::XSoapObject {
private:
    void init();
    int Topology;
    int Dimensions_Levels;
    std::string Reference;
    xsoap::SoapCppArray<ConstraintObject> *App_Constraint_list;
    xsoap::SoapCppArray<DimensionObject> *Dim_C_data;
    ...
}
```

Figure 4.7: XCAT generated AARTObject

retrieving the Virtual Machine, like the original test case. The scheduling is not done and no application is launched. This is because the PPS Binding Phase has not been built and the Mapper doesn't exist yet in GrADSoft. We will now describe the process that was used in this test case.

The Jython script used as XCAT creation mechanism should be written and the four components should be declared in the script. The ports corresponding to these components have to be declared too in this script. When the script is launched, the components are created and they can start their execution.

The Controller component reads the configuration information from the configuration file specified by the user. The information is in fact made of a list of Attribute objects. This list is then transformed by the Adaptor into a SoapCppArray that can be transported between components.

The Controller initializes the Repository and the Builder with the previous SoapCppArray using respectively the *pull* and *builder* port interfaces.

The Controller calls the remote $GetAART()$ function provided by the Builder to retrieve the *example1* AART. The Builder component will here call its own $execute()$ function. This function will make the connection to the Repository component's provides-port and call the $getAART()$ function that it provides. The Repository component then calls the $getAART()$ function provided by the repository object of GrADSoft. This function creates the AART and returns it. The Repository transforms this AART into an AARTObject using the Adaptor and sends it back to the Builder.

As the Builder should not send an empty AART to the Controller, it tests the received AARTObject's value. If the value is null, the Builder will call the $BuildAART()$ function provided by the builder object of GrADSoft. This function builds an AART knowing its name (*example1* or *example2* are the only two possibilities). Once the AART is built, it is transformed into an AARTObject using the Adaptor and sent back to the Controller.

The Controller then calls the remote $GetBestVM()$ function provided by the Resource Selector component to retrieve the best Virtual Machine for the problem. The arguments passed to the $GetBestVM()$ function are the previous AARTObject, the SoapCppArray containing the Attributes' list and a SoapCppArray corresponding to a *VirtualMachineObject* (the XCAT version of a Virtual Machine).

The Resource Selector component has now all the information it needs to call the $GetBestVM()$ function provided by GrADSoft. It just has to transform the received information into compatible GrADSoft type using the Adaptor. GrADSoft modifies the Virtual Machine it was given. The Resource Selector transforms everything back to XCAT types and sends the modified VirtualMachineObject back to the Controller. The Controller finally tests that there was no problem with the Virtual Machine allocation.

As the components act as **services**, they don't shut down at the end of the test case. The user should do that manually. The user should also exit manually from the application manager.

4.6 Problems encountered using XCAT

Although XCAT was released (it it currently used by the NCSA), it can still be considered in test phase, as it is used in a constantly evolving research domain where software packages are continuously upgraded. In fact, all the cases weren't tested yet. The XCAT compiler is mainly used to generate C++ code for much simpler interfaces than those corresponding to this project. More precisely nobody ever used XCAT with nested objects, or nested arrays. This project used a mix of nested objects and arrays, and the objects that had to be transferred between components were much bigger and complicated than ever before. So a few problems were encountered while using the XCAT compiler. These problems were solved either by modifying directly the generated code or by telling the person in charge of the XCAT project what the problem was so that he could apply changes definitively.

4.6.1 The XML mapping files

The XML mapping files have to be written by the user. While creating them, the mapping corresponding to an array of string (string[ ]) was forgotten in the XML files. This generated errors on the receiving side while trying to parse the XML file sent on the wire. This problem has been fixed by adding this mapping.

4.6.2 Boolean

Boolean are used very rarely by C++ programmers. So booleans were not thought as useful in XCAT. As a result, XCAT does not provide boolean as a primitive statement. But GrADSoft uses lots of booleans. So XCAT had to be improved to make boolean as usable as integer.

4.6.3 Inheritance

If there is an object of type *ExtendedObject* that inherits from another object of type *Object*, the code generation done by XCAT for the *ExtendedObject* is not going to be correct.

If we go more into details, while trying to pack the *ExtendedObject* object, the $pack()$ function forgets to pack the attributes from the main class. And while trying to unpack the *ExtendedObject* object, the $unpack()$ function forgets to unpack *Object*'s attributes.

This problem has been solved by adding manually the missing code. This is only a temporary solution and it will be definitively fixed in the Open Framework CGT.

4.6.4 Nested objects

Number/Name of parameters If more than two parameters are declared for a function offered by the provides-port and if we are in the case of a nested object, the compiler mixes the parameters. Indeed, the compiler declares one parameter for each function's argument; these parameters are called $parameter0$, $parameter1$,... In our case $parameter0$ was declared several times but $parameter3$ and $parameter4$ were not declared.

Multiple declaration In the Resource Selector interface, there are two main functions : $GetBestVM()$ and $GetBestVMs()$, which correspond to the GrADSoft functions having the same name. $GetBestVM()$ takes five arguments and $GetBestVMs()$ three. The second/fourth argument of $GetBestVM()/GetBestVMs()$ is an array. There is a string ($expectedLocalName$) on the unpack side that tells the type of the expected variable. So in the generated code, for the arguments cited earlier this string should have the value $Array$. But the string is declared twice, once for each array, which creates compilation problems.

We solved these problems by manually modifying the generated code. Code generation for nested objects will be improved in the Open Framework CGT to suppress these problems.

Conclusion

In this dissertation, we described the goals of the GrADS project and its system architecture. We then introduced GrADSoft, a prototype implementation of the GrADS system. This prototype implemented a number of GrADSoft components but did not yet support distributed execution of these components. Therefore, the goal of this project was to create a distributed version of GrADSoft using XCAT, a component-based framework for building distributed systems.

We then describe the design and process of creating an XCAT-enabled version of GrADSoft. This implementation has been tested and shown that distributed GrADSoft components can connect together and send/receive coherent information. Furthermore, given that the C++ implementation of XCAT is also a prototype, our effort to componentize GrADSoft exposed some limitations to its implementation which provided useful feedback to its developers.

Future work could include adding a more generic Controller component. This component could be an Application Manager Factory, which would produce specific application managers. Also, GrADSoft will continue to evolve, therefore future efforts will also be needed to keep GrADSoft XCAT-enabled.

Appendix

A Glossary

AART (Application Abstract Resource and Topology Model) Cf. Part II Section 1.3.

AppLeS (Application-Level Scheduling) Provides mechanisms that perform scheduling and decision making on behalf of the user.

CoG Kit (Commodity Grid Kit) A Commodity Grid Kit defines and implements a set of general components that map Grid functionality into a commodity environment (cf. Part I Section 1.3).

DOE (Department Of Energy) American Department of Energy.

Framework (Component) It is a specific implementation of a component architecture.

GASS (Global Access to Secondary Storage) It simplifies the porting and running of applications that use file I/O to the Globus environment.

GIIS (Grid Index Information Service) The GIIS provides a means of knitting together arbitrary GIS services to provide a coherent system image that can be explored or searched by grid applications. GIIS provides a mechanism for identifying "interesting" resources, where "interesting" can be defined arbitrarily.

GIS (Grid Information Service) The GIS contains information about the state of the Grid infrastructure. GIS is a general term that includes the Metacomputing Directory Service (MDS). GIS and MDS are often used interchangeably; definitions are still being resolved.

GRAM (Globus Resource Allocation Manager) It processes the requests for resources for remote application execution, allocates the required resources, and manages the active jobs.

GSI (Grid Security Infrastructure) GSI Enables secure authentication and communication over an open network.

GSIFTP (Grid Security Infrastructure File Transfer Protocol) It is essentially a standard FTP enhanced to use GSI security.

HBM (HeartBeat Monitor) It is designed to provide a simple, highly reliable mechanism for monitoring the state of processes. It detects and reports the failure of processes.

HTTPS (Secure HyperText Transfer Protocol) Opens a secure hypertext transfer session with the specified site address.

IR Code (Intermediate Representation Code) Cf. Part II Section 1.3.

LDAP (Lightweight Directory Access Protocol) Used for accessing information about hardware, software, and status in a networked environment.

MDS (Metacomputing Directory Service) It provides a framework for managing static and dynamic information about the status of a computational grid and all its components : networks, compute nodes, storage systems, instruments.

Nexus Nexus is a portable library providing the multithreaded communication facilities required to implement advanced languages, libraries, and applications in heterogeneous parallel and distributed computing environments.

NWS (Network Weather Service) Cf. Part II Section 2.1.1.

PES (Program Execution System) GrADS stage that takes care of the program's execution (cf. Part II Section 1.4.2).

PPS (Program Preparation System) GrADS stage that takes care of preparing the executable before the scheduling phase can take place (cf. Part II Section 1.4.1).

PSE (Problem Solving Environment) A PSE is an integrated collection of software tools that facilitates problem-solving in some domain.

RSL (Resource Specification Language) A language that can be used to specify resource requirements, such as what executable(s) to use, arguments to pass, ...

SIDL (Scientific Interface Definition Language) A specification language.

SWIG (Simplified Wrapper and Interface Generator) It is a software development tool that connects programs written in C, C++, and Objective-C with a variety of high-level programming languages.

Third-party transfer Way of transferring any file from a remote machine to a remote machine.

XML (eXtended Markup Language) XML is a universal format for structured documents and data on the Web.

B Generated files for XCAT

The following 5 files correspond to the object TestObject
TestObject.h
TestObjectPack.cpp
TestObjectPack.h
TestObjectUnpack.cpp
TestObjectUnpack.h

These files corespond to seventeen files generated for the Echo port
ProvidesEcho.h
ProvidesEchoImpl.cpp
ProvidesEchoImpl.h
EchoSkeletonFactory.h
EchoStubFactory.h
EchoTypeFactory.h
Echo_SoapSkel.cpp
Echo_SoapSkel.h
Echo_SoapStub.cpp
Echo_SoapStub.h
Echo_idl.h
UserObjects.h
UserObjectsPack.h
UserObjectsUnpack.h
UsesEcho.h
UsesEchoImpl.cpp
UsesEchoImpl.h

C Application Manager script

This script is the creation service in which the components Generator and Printer are created and connected together.

```
import sys
import cca

from xcat.framework.util import EnvObj
from java.lang import String, Object
from jarray import zeros

# pack the environment object
# this will not be needed if xml is used
providesComponent = EnvObj()
providesComponent.put("component-type", "cpp")
providesComponent.put("exec-dir", "/u/nreycene/Projets/Printer")
providesComponent.put("exec-name", "GRADS_PRINTER_SunOS")

usesComponent = EnvObj()
usesComponent.put("component-type", "cpp")
usesComponent.put("exec-dir", "/u/nreycene/Projets/Generator")
usesComponent.put("exec-name", "GRADS_GENRATOR_SunOS")

# create component wrappers
provides = cca.createComponent(providesComponent)
uses = cca.createComponent(usesComponent)
print "Created component wrappers"

# assign a machine name
cca.setMachineName(provides, "hunk.extreme.indiana.edu")
cca.setMachineName(uses, "hunk.extreme.indiana.edu")
print "Set machine names"

# set a creation mechanism
cca.setCreationMechanism(provides, "gram")
cca.setCreationMechanism(uses, "gram")
print "Set creation mechanism"

# create live instances
cca.createInstance(provides)
print "After instantiation provides"
cca.createInstance(uses)
print "After instantiation uses"
```

```
# connect their ports
cca.connectPorts(uses, "outputEchoPort", provides, "inputEchoPort")
print "Connecting ports"
```

D GrADSoft test case example

The following example is a test case for GrADSoft.

```
/*
 * Example ApplicationManager
 *
 */

#include "gradsoft.h"

/**
 * example appManager1.cc
 *
 * Usage: Once made, you can just call
 * -      <code>appManager1</code>
 *
 * If you want to specify a configuration file use
 * -      <code>appManager1 <configFile></code>
 *
 * The default configFile is called example1.config and contains the
 * following entries:
 * <code>
 * - user.mygrid = dralion.ucsd.edu , soleil.ucsd.edu, o.ucsd.edu,
 *                 torc0.cs.utk.edu
 * - user.num_procs = 3
 * - mds.mds_server = grads.isi.edu
 * - mds.mds_port = 3890
 * - mds.mds_updateBehavior = never
 * </code>
 *
 */

int main(int argc, char **argv) {

  string configFileName = "example1.config";
  list<Attribute*> inConfigList;    // collect this from file
  Builder * newBuilder;
  Repository * newRepository;
  AART * aart1;
  ResourceSelector * rsPtr;
  VirtualMachine ** vmArrayPtr;

  if( argc >= 2 ) {
```

```
    configFileName = string(argv[1]);
  }
  cout << "\nAppManager1: Using configFile: " << configFileName;

  // Set up output logging:
#if 0
  // DEBUGGING: the following will save all gradsoft debugging
  // messages in /tmp/grads.log
  cout << "\nAppManager1: Sending debug, info, and error messages to ";
  cout << "/tmp/grads.log";
  Syslog::setVerbosity(DEBUG);
  Syslog::setOutputTo("/tmp/grads.log");
#endif
  cout << "\nAppManager1: error messages sent to stdout";
  Syslog::setVerbosity(ERROR);

  // Read system config info from #configFileName and place in
  // #inConfigList.
  inConfigList = getConfigInfo(configFileName);
  cout << "\nAppManager1: Read " << inConfigList.size()
             << " config attributes from " << configFileName;

  // DEBUGGING - print out configuration attributes
#if 0
  for( list<Attribute *>::const_iterator it = inConfigList.begin();
       it != inConfigList.end();
       it++ ) {
    cout << "\n";
    ((DiscreteAttribute*) (*it))->TerseDump();
  }
  cout << "\n";
#endif

  // Create a new repository and builder.  Temporary solution so we
  // can retrieve an AART from the builder.  Eventually these should
  // be services which are already available and we just contact them,
  // rather than creating them.
  newRepository = new Repository(inConfigList);
  newBuilder = new Builder(newRepository, inConfigList);

  // Build AART with following characteristics:
  //        * 2-dim mesh topology request
  //        * dimension 0 should be ~3x bigger than dimension 1
```

```
  //        * all-to-all communication > 10 Mb
  //        * preference for PII processors
  //        * preference for scalapack software
  cout << "\nAppManager1: Building AART.";
  string aartName = "example1";         // unique AART identifier
  aart1 = newBuilder->GetAART(aartName);

  // DEBUGGING - print out the retrieved AART
#if 0
  cout << "\n"; aart1->Dump();  cout << "\n";
#endif

  // Create a new resource selector.  Temporary solution - eventually
  // resource selector should be a service which is already available
  // and we just contact it (without having to create it first ;-)
  rsPtr = new ResourceSelector();

  // Request Virtual Machine from Resource Selector
  // Exit program if any errors are found since rest of example
  // program will fail.
  cout << "\nAppManager1: Calling GetBestVM to retrieve VM...";
  vmArrayPtr = new VirtualMachine*[1];
  if( (rsPtr->GetBestVM(aart1, inConfigList, vmArrayPtr)) != 0) {
    cout << "\nAppManager1: Error!  Problem with GetBestVM.\n\n";
    return 1;   // error
  }
  if( vmArrayPtr == NULL ) {
    cout << "\nAppManager1: Error!  Problem with vmArrayPtr\n\n";
    return 1;   // error
  }
  if( vmArrayPtr[0] == NULL ) {
    cout << "\nAppManager1: Error!  Problem with vm allocation\n\n";
    return 1;   // error
  }

  // CHECK RETRIEVED VM
  cout << "\n\nAppManager1: Dumping retrieved virtual machine.";
  (vmArrayPtr[0])->Dump();

  // TODO: the appropriate next step is for the appManager to actuate
  // the application itself or call an actuator.  However, we do not
  // currently have an actuator.  Until an actuator is built testing
  // of the system can be achieved using a shell script that parses
```

```
  // the output of the virtual machine and then actuates the
  // application from the shell script.
  // Or course other steps are missing as well (monitoring, contract
  // development, a functional backend to the builder and repository,
  // etc).  So much to do ...

  cout << "\nAppManager1: done.\n";
}
```

Bibliography

[1] CORBA, visited 01-03-2002. http://www.omg.org/gettingstarted/corbafaq.htm.

[2] LDAP, Lightweight Directory Access Protocol, visited 01-03-2002. http://www.openldap.org/.

[3] Network Weather Service, visited 01-03-2002. http://nws.npaci.edu/.

[4] Tcl/Tk, visited 01-03-2002. http://www.tcl-tk.net/.

[5] The GriPhyN Project, visited 01-03-2002. http://www.griphyn.org/.

[6] MzScheme, visited 10-02-2001. http://www.cs.rice.edu/CS/PLT/packages/mzscheme/.

[7] WxPython, visited 10-02-2001. http://www.wxpython.org/.

[8] Guile, visited 10-03-2001. http://www.gnu.org/software/guile/guile.html.

[9] Objective C, visited 10-03-2001. http://www.objective-c.org/.

[10] GrADS Project Web Page, visited 11-06-2001. http://gridlab.ucsd.edu/~ grads/.

[11] OMG-IDL, visited 12-06-2001. http://www.omg.org/gettingstarted/omg_idl.htm.

[12] XML, visited 12-06-2001. http://www.w3.org/XML/.

[13] XML Schema, visited 12-06-2001. http://www.w3.org/XML/Schema.

[14] Jython, visited 9-10-2001. http://www.jython.org.

[15] SWIG, visited 9-10-2001. http://www.swig.org.

[16] Ian Foster ans Carl Kesselman. Globus: A Metacomputing Infrastructure Toolkit.

[17] Ian Foster ans Carl Kesselman and Steven Tuecke. The Nexus approach to integrating multithreading and communication. In *Journal of Parallel and Distibuted Computing*, 1996.

[18] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinski. Toward a Common Component Architecture for High-Performance Scientific Computing. In *Proceedings of the Eighth IEEE International Symposium on High Performance Distributed Computing Conference, Redondo Beach, California*, August 3-6 1999.

[19] Grid Computing Laboratory at the University of California San Diego. AppLeS, visited 01-03-2002. http://apples.ucsd.edu/.

[20] Francine Berman, Andrew Chien, Keith Cooper, Jack Dongarra, Ian Foster, Dennis Gannon, Lennart Johnsson, Ken Kennedy, Carl Kesselman, John Mellor-Crummey, Dan Reed, Linda Torczon, and Rich Wolski. The GrADS Project: Software Support for High-Level Grid Application Development, July 30 2001.

[21] Randall Bramley, Kenneth Chiu, Shridhar Diwan, Dennis Gannon, Madhusudhan Govindaraju, Nirmal Mukhi, Benjamin Temko, and Madhuri Yechuri. A Component Based Services Architecture for Building Distributed Applications. In *Proceedings of Ninth IEEE International Symposium on High Performance Distributed Computing Conference, Pittsburgh*, August 1-4 2000.

[22] Randall Bramley, Dennis Gannon, Madhusudhan Govindaraju, and Aleksander Slominski. XSOAP, April 2001.

[23] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. WSDL, visited 12-06-2001. http://www.w3.org/TR/wsdl.

[24] Holly Dail. Holly Dail Masters Thesis, 2002.

[25] Andrew Davidson, Matthew Fuchs, Mette Hedin, Mudita Jain, Jari Koistinen, Chris Lloyd, Murray Maloney, and Kelly Schwarzhof. SOX, visited 12-06-2001. http://www.w3.org/TR/NOTE-SOX/.

[26] Jack Dongarra, Antoine Petitet, Mark Mazina, John Mellor-Crummey, Fran Berman, Otto Sievert, and Holly Dail. GrADS Library/Compiler/Scheduler Interaction Scenario, June 9 2000.

[27] ISE (Interactive Software Engineering). Eiffel, visited 10-02-2001. http://www.eiffel.com/.

[28] Jason Novotny et al. MyProxy, visited 09-12-2001. http://dast.nlanr.net/Features/MyProxy/.

[29] Extreme Computing Lab. XCAT Java, visited 01-03-2001. http://www.extreme.indiana.edu/xcat/.

[30] Extreme Computing Lab. XCAT, visited 01-03-2002. http://www.extreme.indiana.edu/xcat/.

[31] Extreme Computing Lab. Active Notebook, visited 9-25-2001. http://www.extreme.indiana.edu/an/.

[32] European Center for Medium Range Weather Forecasting (ECMWF). Integrated Forecasting System, visited 01-03-2002. http://www.ecmwf.int/research/ifsdocs.

[33] Ian Foster and Carl Kesselman. The Globus Project: A Status Report. In *Proc. IPPS/SPDP '98 Heterogeneous Computing Workshop*, 1998.

[34] Ian Foster and Carl Kesselman. *The GRID: Blueprint for a New Computing Infrastructure.* Morgan-Kaufmann, 1998.

[35] Ian Foster, Carl Kesselman, and Steven Tuecke. The Anatomy of the Grid, Enabling Scalable Virtual Organizations, 2001.

[36] Java Grande. Computing Portals, visited 9-2-2001. http://www.computingportals.org.

[37] Argonne National Laboratory Gregor von Laszewski. Distributed Programming in Computational Grids With Java.