
An XCAT Based Component Architecture for GrADSoft

ISTY 3 year 2001–2002

Version 2.0

January 18, 2002

Authors

Caroline OLARIU
Nicolas REY-CENEVAZ

Contents

1	GrADSoft	2
1.1	Introduction	2
1.1.1	Prerequisites	2
1.1.2	Files description	3
1.2	GrADSoft architecture	3
1.2.1	The functionalities/services provided by each module	4
1.2.2	GrADSoft status	6
1.3	A scenario explaining GrADSoft behavior	6
1.4	Application Manager behavior	11
2	XCAT Components	12
2.1	Open Framework Code Generation Toolkit	12
2.1.1	Introduction	12
2.1.2	Architecture	12
2.1.2.1	Interface Specification	13
2.1.2.2	Code Template	14
2.1.2.3	Type Mappings	14
2.2	XSOAP	14
2.2.1	Introduction	14
2.2.2	RMI	15
2.2.3	XSOAP-Java	15
2.2.4	XSOAP-C++	15
2.3	Common Component Architecture	17
2.3.1	Introduction to the Common Component Architecture	17
2.3.2	Port	17
2.3.2.1	Provides-Port	18
2.3.2.2	Uses-Port	18
2.3.2.3	PortInfo	18
2.3.3	SIDL translation	19
2.3.4	The Services object	19
2.3.5	The Component interface	21
2.3.6	Application Manager	22

Appendix	23
A Generated files for XCAT	23
B Application Manager script	24
C GrADSoft test case example	26
Bibliography	29

List of Figures

- 1.1 Functional relationship diagram 4
- 1.2 UML scenario of GrADSoft execution [17] 10
- 1.3 Configuration file example 11

- 2.1 Architecture of the Open Framework Code Generation Toolkit 13
- 2.2 Architecture of the XML Schema Document Generator in the Open Framework CGT 14
- 2.3 Java RMI Architecture 15
- 2.4 SoapRMI 1.1 Java Architecture 16
- 2.5 SoapRMI 1.1 C++ Architecture 16
- 2.6 SIDL specification of PortInfo 18
- 2.7 Echo Interface 19
- 2.8 String mapping in XML 19
- 2.9 Services Interface 20
- 2.10 ConnectionEvent 21
- 2.11 ComponentID Interface 21
- 2.12 Component Interface 21
- 2.13 TestPrinterComponent 22
- 2.14 TestGeneratorComponent 22

List of Tables

- 1.1 The GrADSoft files organization 3
- 1.2 The GrADSoft interface 7

Chapter 1

GrADSoft

1.1 Introduction

GrADSoft [19], [13] is a software project that aims to design a software architecture which is a prototype for the GrADS project. GrADSoft was developed in concert with the GrADS project and follows GrADS specifications. It provides a management framework for application and resource information. Various GrADS modules (such as resource selectors, application monitors, compilers, performance contracts, libraries, problem solving environments, ...) can interact with each other via this framework. GrADSoft should facilitate the compilation, execution, and monitoring of distributed applications.

GrADSoft should have the capability to launch an application both as a unique executable running on a single machine or as different executables running as independent, distributed modules (acting as services). GrADSoft is written in C and C++, and has been compiled successfully under Linux, HP-UX and Sun Solaris.

GrADSoft design methodology is object oriented, which is the first step towards a distributed design. This methodology also allows easier addition of new techniques and features, such as dynamic contract renegotiation and more software-controlled program preparation work. The current status of the GrADSoft effort (a first prototype was released) does not achieve all of the long-term GrADSoft goals. Meanwhile, it does provide many of the necessary basic functions and is a useful and usable proof of concept.

1.1.1 Prerequisites

GrADSoft has been developed for Unix and its variants (HP-UX, Solaris, Linux, ...). Several software packages need to be installed in order to run GrADSoft. These packages are :

- LDAP (Lightweight Directory Access Protocol) :
GrADSoft has been built on top of the OpenLDAP package, which is an open source implementation of LDAP [2]. LDAP is a protocol for accessing online directory services. It runs directly over TCP.
- NWS (Network Weather Service) :
The NWS [3] is a distributed system that monitors and forecasts the performance of various network and computational resources. The service operates a distributed set of performance sensors (network

monitors, CPU monitors, etc.) from which it gathers readings of the instantaneous conditions. It then uses numerical models to generate short term forecasts of resource availability. NWS use dynamic schedulers.

- AppleSeeds :
AppleSeeds [9] is a package of useful utilities, from UCSD. AppleSeeds is composed of a set of utilities intended to ease the development of distributed schedulers.
- Perl :
The standard eclectic language (version 5.0 or higher).
- GNU make
- GNU autoconf
- Doxygen [22] :
This package is used to build HTML documentation from source code. It is not essential to basic operations.
- Graphviz [21] :
This package is used to create graphics. It is not essential to basic operations, but is used by Doxygen.

1.1.2 Files description

The GrADSoft files are organized following the directories described in Table 1.1.

Directory Name	Description
doc	Contains documentation (primarily software architecture design documents)
doc/html	Directory containing dynamically generated source code documentation (optional)
include	Common (exported interface) include files
aart	Source code for the AART module
builder	Source code for the builder module
misc	Source code for common classes used by many modules
repository	Source code for the repository module
resourceSelector	Source code for the resource selector module
virtualMachine	Source code for the virtual machine and resource modules
srlink	A directory that contains soft links to all the individual source code files (optional)
test	Source code for module testers. Examining the tester is probably the best way to understand the usage of each module
examples	Directory containing examples of how you can use this code

Table 1.1: The GrADSoft files organization

1.2 GrADSoft architecture

GrADSoft is organized in different modules interacting together. Those modules are part of a single executable and so can only be executed on a single machine. As GrADSoft was built to be a prototype of the GrADS effort, its architecture is as similar as possible to the GrADS specifications. The interface chosen to implement this architecture is presented in Section 1.2.2. But before that a more detailed explanation of each part of the software can be found in Section 1.2.1 as well as a description of the services they provide.

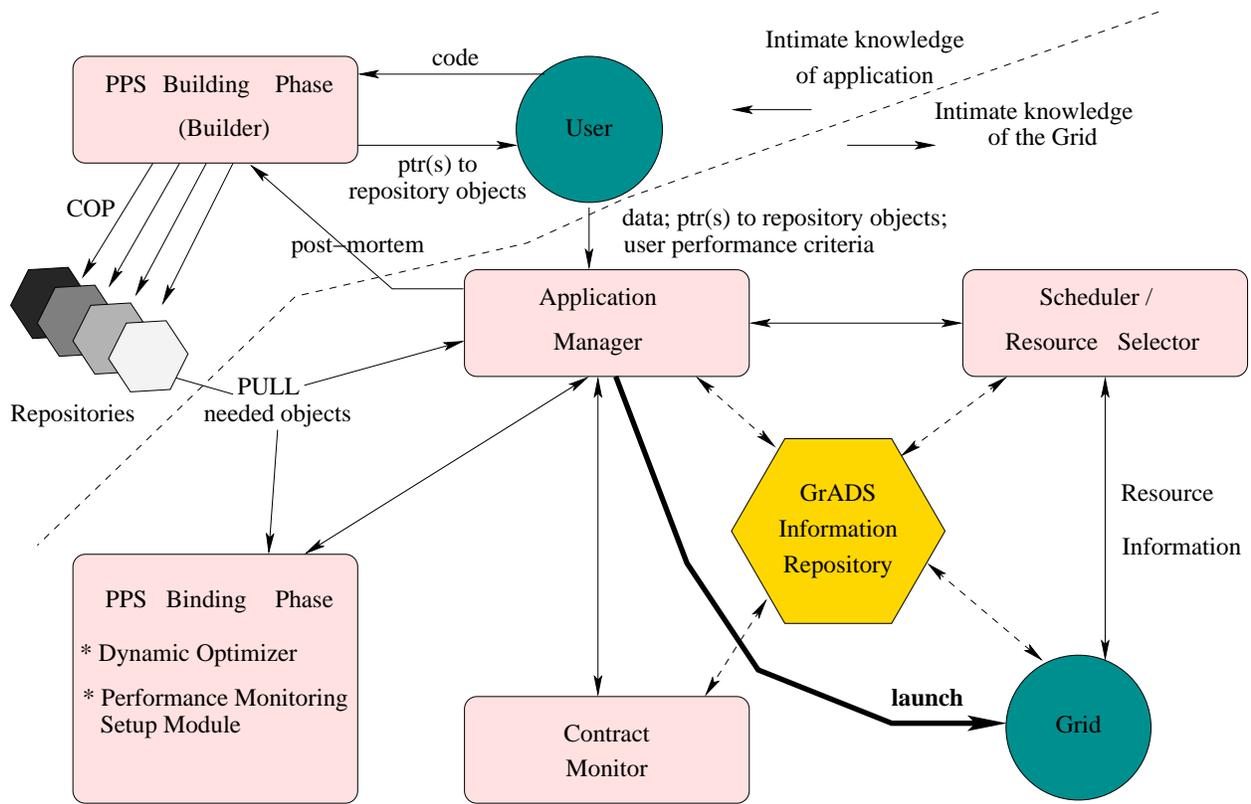


Figure 1.1: Functional relationship diagram

1.2.1 The functionalities/services provided by each module

We will now explain in detail what each module of GrADSsoft shown in Figure 1.1 does. We will also specify its role in the preparation and execution processes.

PPS Building Phase : the Builder The Builder behaves as a compiler. In today's GrADSsoft prototype, the Builder creates the following objects :

- AART Model object
- IR Code object
- Mapper object
- Resource Selection Evaluator object

In the future, the Builder will not *create* these objects anymore. It will only *return pointers to existing repository objects*, as it should do in the GrADS theory.

The four objects mentioned above compose the COP (Configurable Object Program). Unlike its composing objects that can be stored in repositories, the COP doesn't have persistence.

We should now detail what is the role of each of these objects composing the COP :

- AART Model object :

An AART Model object encapsulates application characteristics and requirements. Three fundamental features of an AART Model are :

- The AART Model describes the resources and topologies desired by an application, independently of any given run-time data. For example, the AART Model contains the kind of resource topology (e.g., machines are arranged in a mesh) without mention of the exact size of this mesh (which is highly data and resource dependent).
- The AART Model characteristics may change based on the problem size. So the model may be a continuum of models or a discrete number of very different models based on that size. Below some application-specific problem size, an AART Model will likely specify using a single compute resource at one location.
- The AART Model attempts to describe resources and topologies necessary to execute an "efficient computation".

The topology-type characteristic will typically have a dimensionality (for example if the topology is a mesh, it can be a 3-D mesh) or a number of levels (when then chosen topology is a tree for example).

The current AART Model is designed to describe characteristics at both the total-program level and characteristics that apply to specific dimensions or levels of the topology. Because of this, a topology-type characteristic also has a role as a meta-characteristic.

- IR Code object :

The IR Code object contains the application's code as it is created by the Builder (before the PPS Binding phase).

- Mapper object :

The Mapper is not fully instantiated until the resource selection process is complete. This object will initially consist of methods to do a mapping on the selected resources given the AART Model, Virtual Machine, plus meta-data about the input data (the input data usually consists only of the size of the problem).

The Mapper can decide to change the work allocation that was previously chosen for efficiency. But it can't change the Virtual Machine. So, if the work allocation is too "big", the Resource Selector needs to be called again.

- Resource Selection Evaluator object :

The Resource Selection Evaluator (RSE) object is not fully instantiated until the Mapper is completely built. This object will initially consist of methods to create an evaluation metric given the AART Model, the Virtual Machine, and the Mapper.

The Resource Selection Evaluator will be the PPS component that accepts feedback from the contract monitoring system.

Application Manager The Application Manager can be thought of as the user interface. It acquires input data and other run-specific parameters from the user. The Application Manager coordinates the components' activities by storing and passing around pointers to objects and by invoking object methods (cf. Section 1.3 for the sequence of operations the Application Manager does). The Application Manager may be unique for each application.

Scheduler/Resource Selector The Resource Selector collects information on the current Grid environment state from the GIS. It uses the AART Model expanded with information on input-data and parameters in order to select resources. The Resource Selector returns one or more Virtual Machines and a suggested work allocation, but it is the Mapper that is responsible for the final data layout.

PPS Binding Phase (Dynamic Optimizer and Performance Monitoring Setup Module) The PPS Binding Phase is composed of two modules, the Dynamic Optimizer, which queries the target machines for configuration data and transforms the object program into an optimized executable that can now run efficiently on the target machines; and the Performance Monitoring Setup Module which inserts in the program the sensors and actuators needed by the runtime system.

GrADS Information Repository The GrADS Information Repository holds information about the Grid environment, checkpoints of the application at each execution stage, the Virtual Machine for the problem run, the Contract Monitor's output, and meta-data about the execution.

Contract Monitor The Contract Monitor follows the execution of the application by monitoring the output of the sensors placed in the finalized and optimized executables.

Launcher The Launcher loads and starts the application on the Virtual Machine. In GrADSoft the Launcher functionality is encapsulated in the Application Manager.

1.2.2 GrADSoft status

Currently, GrADSoft implements only a subset of its architecture. The current status of the GrADSoft interfaces (classes, structures,...) is briefly described in Table 1.2. We note a couple of the differences between implementation and architecture below :

- GrADSoft developers team provided the Repository with the *getAART()* and *putAART()* functions. So the Repository only stores AART objects, whereas the GrADS specification says the Repository should also store the IR Code, the Mapper, and the Resource Selection Evaluator, which are all the objects composing the COP.
- The Builder has limited functionalities. It can only be called by the way of the *GetAART(string & aartReference)* function and it only recognizes two references : the strings *AARTexample1* and *AARTexample2*. So only two predetermined AART objects can be created.

1.3 A scenario explaining GrADSoft behavior

Figure 1.2 shows a UML scenario of GrADSoft execution. Other scenario may be possible. The scenario described below is what we can consider as the *ideal* execution case, where no rescheduling is needed for the application [17].

Name	Type	Description
AART	Class	Application Abstract Resource and Topology (AART) Model
AppRequireInfo	Structure	Used in the Resource Selector
Attribute	Class	General descriptor classes, organized by type
Builder	Class	Used to build Intermediate Code representations and provide AART objects to callers
Constraint	Class	A valued attribute, its negotiability and its importance
Dimension	Structure	Stores information about each dimension in an AART
DiscreteAttribute	Class	The attribute value is a discreteValue
DiscreteValue	Class	A value object that stores a string
IntervalAttribute	Class	The attribute value is an intervalValue
IntervalValue	Class	A value object that holds a numeric interval
L2TypeLDAPInfo	Structure	It is a table that maps certain type values to MDS class
MDSAttribute	Class	Used to manage resource characteristics obtained from the Globus MDS via a GIIS
NWSAttribute	Class	Used to store resource characteristics obtained from NWS
Repository	Class	A repository is designed to provide run-to-run storage of AARTs, mappers, and other long-lived GrADS objects
Resource	Class	A Resource represents a resource in the system
ResourceSelector	Class	Selects application-appropriate resources
SeriesAttribute	Structure	The attribute value is a seriesValue
SeriesValue	Class	A value object that can store a data series and (optionally) characteristics such as time stamps for that data series
Syslog	Class	Utility class to centralize all the logs
UserConfigInfo	Structure	Used in the Resource Selector
VirtualMachine	Class	This class stores information about the target resource arrangement for an application

Table 1.2: The GrADSsoft interface

The user gives its source code or a handle to an existing IR Code object previously created for the user to the Builder. The source code may be annotated with resource selection or performance behavior information.

The Builder uses library references to build the COP. The COP includes the IR Code, AART Model, Mapper and Resource Selection Evaluator. Then the COP is moved to the PPS Repository and a pointer to this location in the PPS Repository is returned to the user. At this point, the Mapper and the Resource Selection Evaluator have too little information to do anything useful.

Then, the user invokes the Application Manager. The Application Manager needs the handle to the COP, I/O location information, the problem size information (specifically information to allow calculation of memory requirements), plus any user preferences (performance metrics and other run-specific parameters desired or required). The Application Manager retrieves the pieces of the COP from the PPS Repository. The AART Model is here combined with the problem run information, resulting in the Resource Selector Seed Model (RSSM). This produces the preliminary state necessary for the Mapper and the Resource Selector to start being useful.

The Resource Selector uses the RSSM to query the GIS for Grid information (CPU utilization, bandwidth, memory, disk space) on available machines. The Resource Selector then develops a feasible Virtual Machine using this state information and the services of the Mapper as well as the Resource Selection Evaluator (RSE). This Virtual Machine is returned to the Application Manager.

The Application Manager gives the COP and the Virtual Machine to the PPS Binding phase and stores the Virtual Machine in the GrADS Information Repository. The GrADS Information Repository contains checkpoints on the problem run (i.e. application and data).

The PPS Binding phase uses the Mapper to map data to the Virtual Machine, and creates optimized binaries for each architecture existing in the Virtual Machine using the Dynamic Optimizer. It also inserts monitoring sensors based on information from the Performance Monitoring Setup Module. Handles to the optimized problem run binaries are passed back to the Application Manager, which again checkpoints its state to the GrADS Information Repository.

The Application Manager starts the Contract Monitor and contacts the Launcher with the optimized executables, the Virtual Machine, the problem information and the final mapping. The Launcher starts the program execution on the Grid.

As the code runs, the Contract Monitor collects data from the sensors. It uses the Performance Contract and contract evaluation method from the Performance Monitoring Setup Module to determine if the application is delivering an acceptable level of performance based on the models. In addition, the Contract Monitor may try to make some determination of the cause of the poor performance.

Moreover, the Contract Monitor output, as well as the original sensor output, can be archived in the GrADS Information Repository for later use to refine models or guide future execution.

At the end of the execution, the Application Manager receives all completion events, the execution's result and performance data. It shuts down the Contract Monitor and stores meta-data about the run in the GrADS Information Repository. The Application Manager notifies the user of the program execution's end.

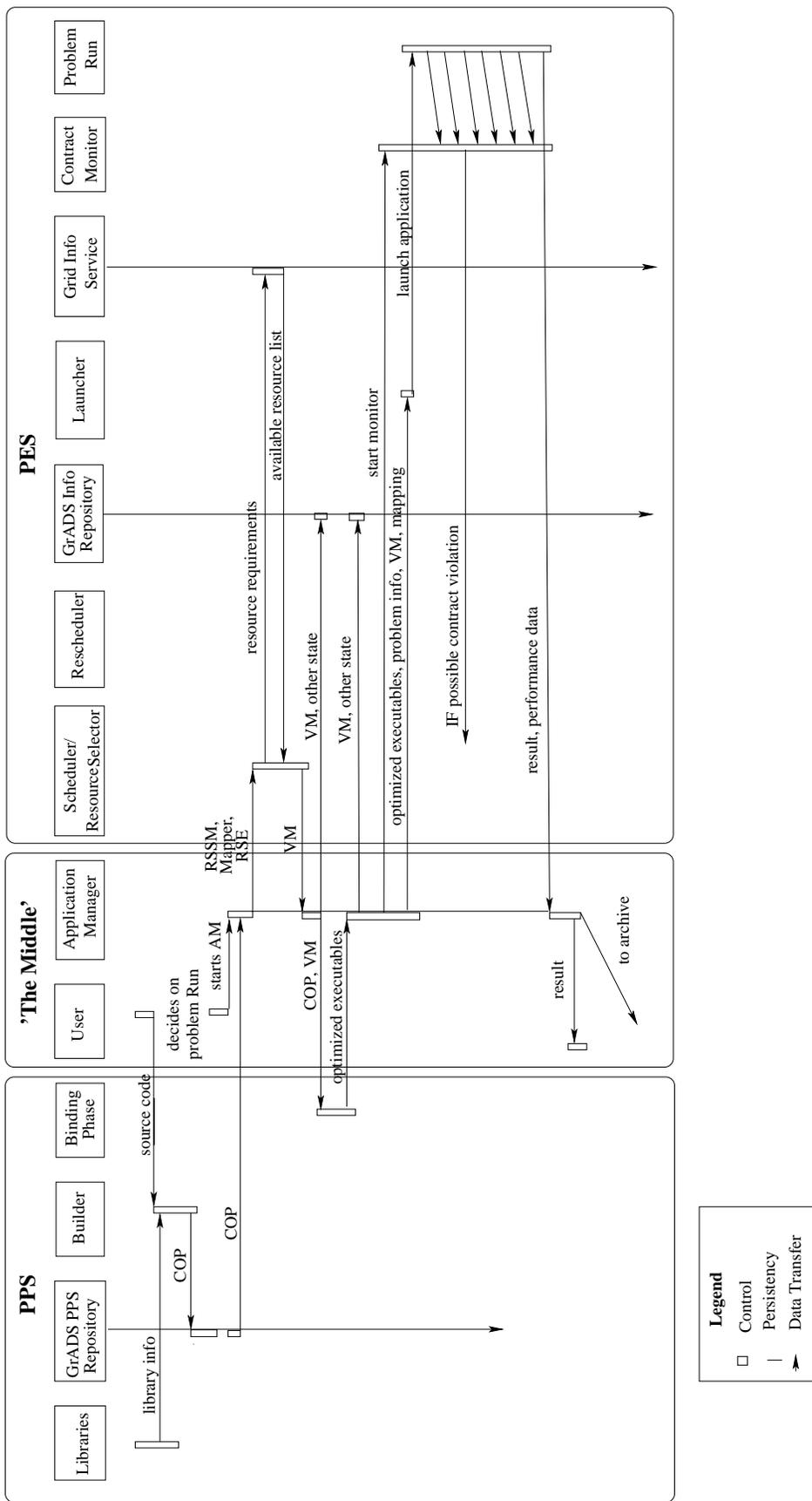


Figure 1.2: UML scenario of GrADSoft execution [17]

1.4 Application Manager behavior

GrADSoft requires an Application Manager. The Application Manager's main task is to coordinate the activities and data flow for the other GrADS modules. Since different applications may want to utilize the GrADS infrastructure in different ways the Application Manager may be different for each application. But each running application must have an Application Manager.

For example, the application manager can do the following :

- read the configuration file written by the user
- call the Builder to retrieve the AART appropriate to the application
- call the Resource Selector to get a Virtual Machine for the current problem
- launch the application

An example of configuration file can be found in Figure 1.3.

```
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
```

Figure 1.3: Configuration file example

Chapter 2

XCAT Components

The idea of using components to deal with the complexity of developing an application is becoming increasingly popular. Components can be thought of as basic units of software that can be put together to form applications. The goal of such systems is to enable programmers to accelerate project development by introducing higher-level abstractions and allowing code reusability, as well as to provide component interfaces which facilitate the task of team interaction. Today software component architecture is a standard in many areas of application development.

2.1 Open Framework Code Generation Toolkit

2.1.1 Introduction

Distributed systems like CORBA [1], DCOM [20], CCAT [18] and XCAT [16] have complex run time systems. These systems are designed with multiple communication protocols, varying QoS (Quality of Service) guarantees, load balancing techniques and interoperability capabilities. The interfaces to these systems are described using a specification language. The common interface languages include OMG-IDL [4], SIDL (Scientific Interface Definition Language), XML [5], XML-Schema [6], SOX [14] and Java classes. In order to use these different systems a user is required to be familiar with the intricate details of the IDLs (Interface Definition Language) and their usage. The Open Framework Code Generation Toolkit (CGT) addresses this problem by providing an IDL that captures the useful features of the various IDLs. It then provides an architecture that enables code generation for different distributed object systems such as those described above. The Open Framework CGT is an XML Schema based software that is designed to work as a code generator for different distributed object systems. The toolkit has been developed in the Extreme Computing Lab at Indiana University.

2.1.2 Architecture

Figure 2.1 shows the architecture of the Open Framework CGT. The toolkit needs three different kinds of input to be able to generate code.

- Interface Specification
- Code Template

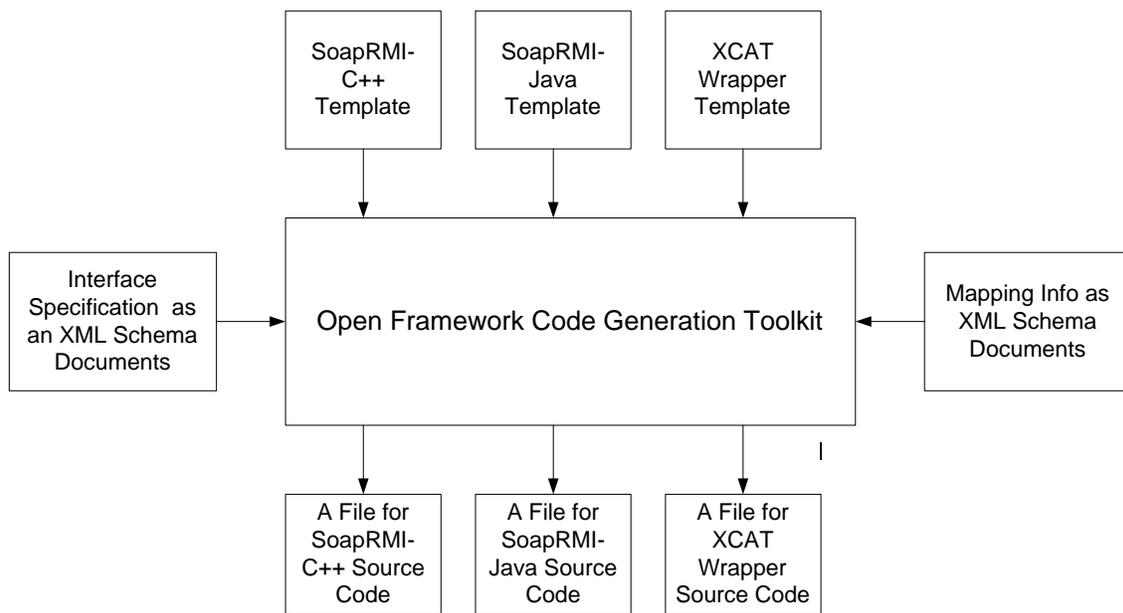


Figure 2.1: Architecture of the Open Framework Code Generation Toolkit

- Type Mappings

All these inputs are specified as XML Schema documents. The code generator has no built-in knowledge about the code it needs to generate. To learn about the format of the code to be generated, the code generator uses the code templates. The code templates describe the format of the expected output code. The code generator just understands the grammar that is used by code templates. The interface specification describes the interface of the distributed system for which code is being generated. The type mappings are used to provide bindings for the various types that may be found in the code templates. For example, the type mappings need to provide a list of primitive and user defined types for the language in which the target code needs to be generated.

2.1.2.1 Interface Specification

Interfaces encapsulate the internal mechanisms of an object. An Interface Definition Language (IDL) allows for describing various interfaces in a system. The Open Framework CGT accepts specification of interfaces as either Java interfaces and classes or XML Schema documents. The XML Schema documents conform to an XML Schema that captures an important subset of the features in WSDL [12].

Figure 2.2 shows the design of a simple XML Schema document generation architecture that is used by the Open Framework CGT. Java classes are provided as input to the schema generator. It then generates an XML Schema document that conforms to a fixed XML Schema. This alleviates the need for the tedious and error prone task of manually writing the XML Schema documents.

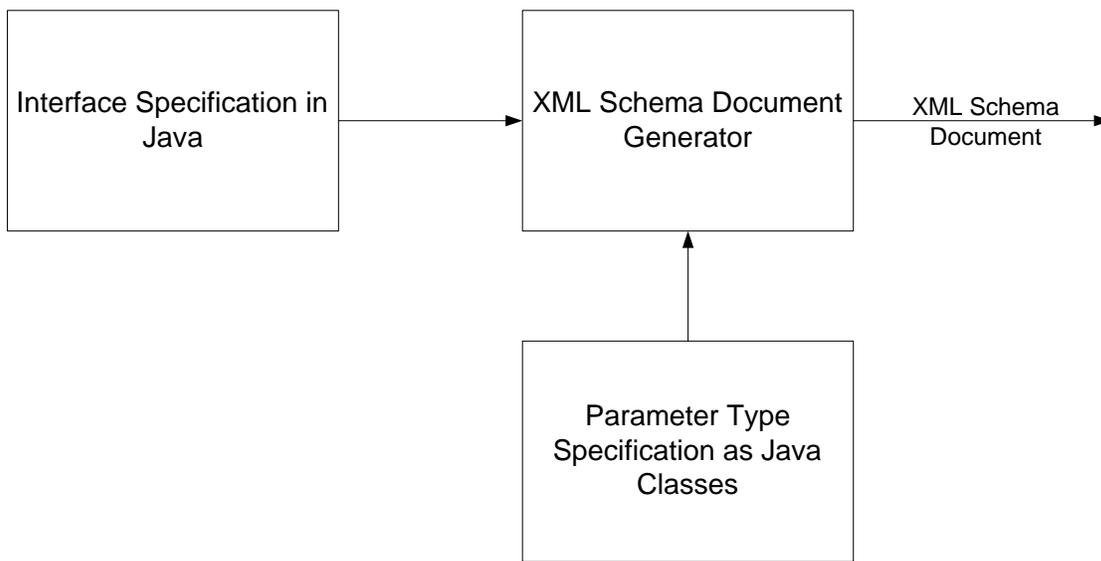


Figure 2.2: Architecture of the XML Schema Document Generator in the Open Framework CGT

2.1.2.2 Code Template

The code templates need to be provided as an XML Schema document that conforms to a fixed grammar. Typically, for a given distributed object system the set of files and the format of each file that needs to be generated is fixed. However the interfaces that applications use vary. This results in a change in the method names and method parameters in the generated code. A code template describes the structure of code that needs to be generated and identifies the parts of the code that are interface specific such as keywords or variables that need to be replaced with a suitable mapping.

2.1.2.3 Type Mappings

The code generation architecture is very flexible as it allows users to steer the process with various mappings. Mappings are used to bind the variables present in code templates. These mappings can be obtained either from the interface specification or can be supplied by the user.

2.2 XSOAP

2.2.1 Introduction

XSOAP is an implementation of the Java RMI API in both C++ and Java that uses SOAP as its wire protocol for communication with remote objects. Java RMI provides an elegant model for communication with objects that reside in different processes. XML has emerged as a promising standard for data representation in a language-independent manner. HTTP is a widely used network protocol that is simple to use. SOAP [11] defines XML based communication and precisely states the protocol for using XML as the data format and HTTP as the network protocol.

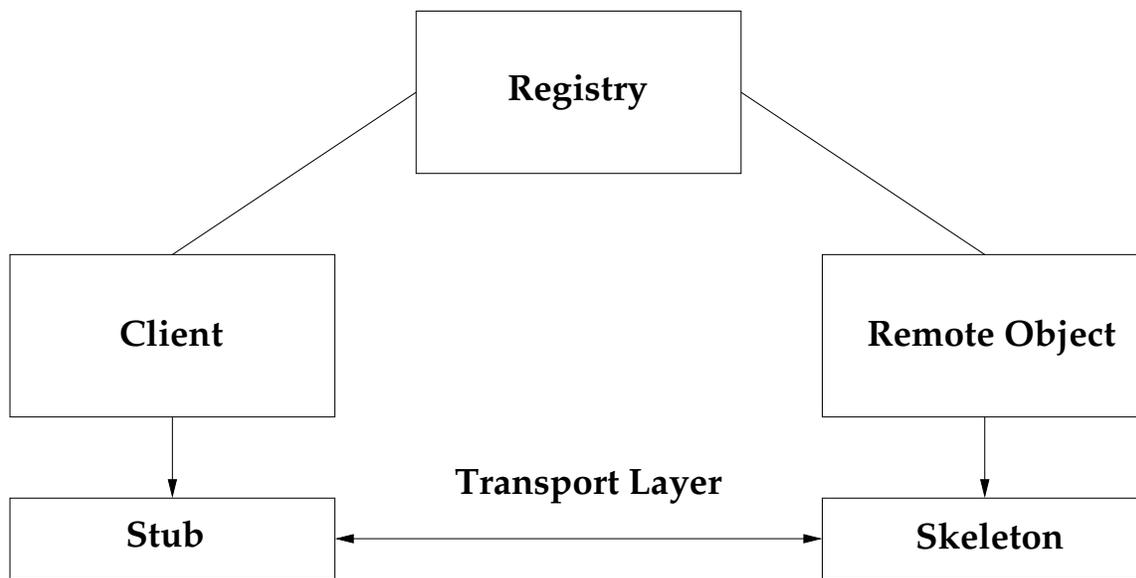


Figure 2.3: Java RMI Architecture

2.2.2 RMI

Java RMI is a framework designed to simplify distributed object oriented computing in Java. An overview of the Java RMI model is shown in Figure 2.3.

RMI is essentially a client-server model. The **stub** acts as a proxy for the remote object. The **skeleton** is an object that lives in the same JVM as the remote object and handles communication with the stub. The **registry** is used to manage remote references. The server binds a remote reference to itself onto the registry. To obtain a remote reference to the server, the client contacts the registry which may be on a different remote host. The client can use the remote reference obtained from the registry to invoke methods on the remote object.

2.2.3 XSOAP-Java

Figure 2.4 represents the architecture of XSOAP-Java framework. Due to its modularity, it is possible to plug-in different implementations of various modules. Every client has access to a *soap-services* module. This module handles all the details of SOAP specific serialization and deserialization of objects. *UnicastRemoteObject* is used to make objects accessible from remote locations. The *dynamic-stub* is used to dispatch serialized objects to the HTTP connection layer. On the receiving end (*embedded-server*), every request is handled by a *dynamic-skeleton*. A mapping layer, *soap-encoding/Mapping*, binds namespaces to interfaces and types in the system.

2.2.4 XSOAP-C++

Figure 2.5 shows the architecture of XSOAP-C++. It is based on the traditional RMI model as it uses stubs and skeletons for every remote interface. The *communication* layer receives messages from the stub and adds the SOAP envelope onto it. The message is then sent to the HTTP layer to be sent on the wire. On

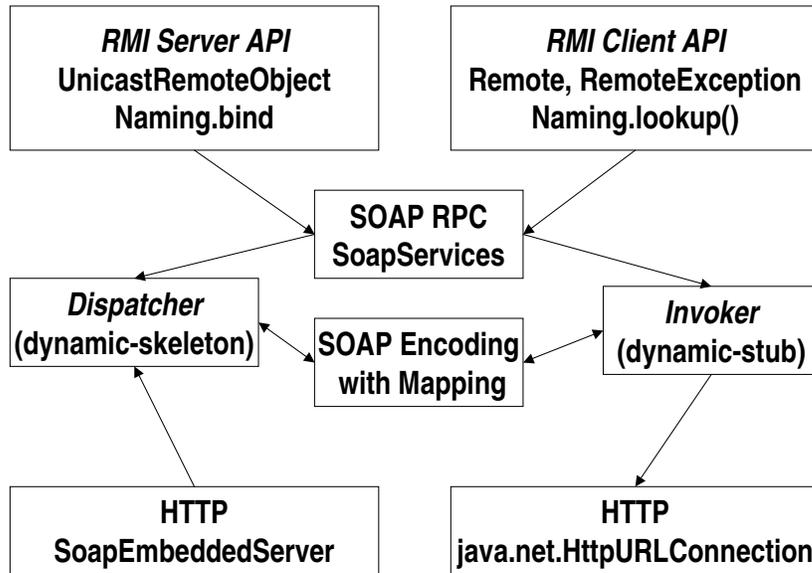


Figure 2.4: SoapRMI 1.1 Java Architecture

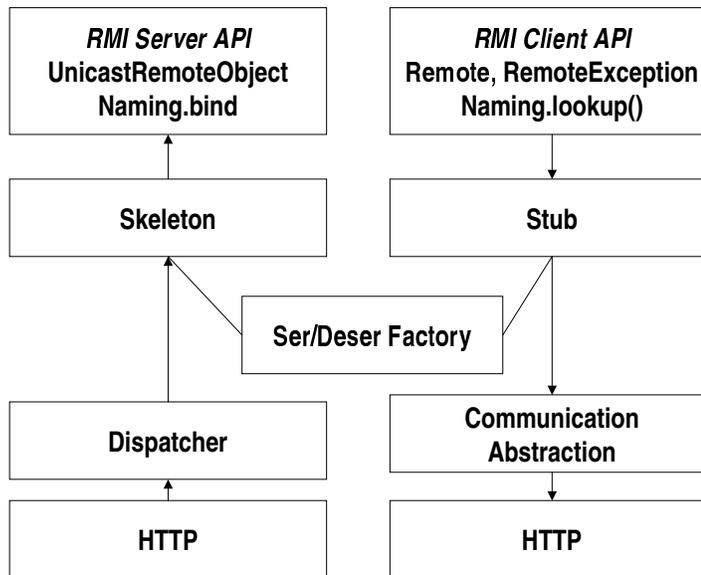


Figure 2.5: SoapRMI 1.1 C++ Architecture

the receiving end a *dispatcher* spawns a new thread for every request that it receives. It then dispatches the call to the appropriate remote object to which the call is directed.

2.3 Common Component Architecture

2.3.1 Introduction to the Common Component Architecture

Recently, representatives from several DOE national laboratories, the University of Utah, Indiana University, and NSCA have defined a specification for a standard component model for distributed and parallel scientific computing called the Common Component Architecture (CCA).

The CCA [8], [10], [23] consists of two type of entities : components, and framework. A framework is a specific implementation of a component architecture, more precisely it is a specific implementation of the CCA specification.

The philosophy of the CCA is to precisely define the rules for constructing components and the specification of the required behavior that a component must exhibit for it to coexist with other components within a CCA framework. A CCA framework is a software environment that allows components to be dynamically instantiated, coupled together, and have methods invoked on them.

Currently, the CCA does not specify how the framework is constructed or how the user interacts with the framework to connect components together. The reason is that there are many different frameworks that can be used in different situations. Some frameworks will be designed to optimize the use of the components that are distributed across a wide-area Grid. Others will be designed to optimize the composition of components that run on a single, massively parallel supercomputer.

CCA components may be written in Java, Fortran, C or C++. It is up to the framework to ensure that the appropriate infrastructure is in place to allow components from different languages to interoperate.

The Extreme Computing Lab has designed and built an implementation of the CCA specification over Globus. This implementation called XCAT (eXtreme Component Architecture Toolkit) [16] has been developed for two target component implementation languages : Java and C++. Since GrADSoft has been written in C++, we will describe in the following sections how the C++ version of XCAT works and how it has been implemented.

2.3.2 Port

The port is a fundamental CCA concept. A port is a public communication interface of a component. A port can be either a provides-port or a uses-port. The characteristics of these two different ports are explained in the next two sections.

2.3.2.1 Provides-Port

A **provides-port** is an interface of functions that the component implements which can be referenced and *used* by other components. A component can have zero or more provides-ports. A provides-port can also be thought of as a *service* that is *provided* to other components or to the framework. The member functions of a provides-port may be thought of as *handler* functions that are executed by the component on behalf of the component's *users*. In some cases, the provides-port interfaces are implemented by the component object, and in other cases the provides interfaces are implemented by another object instantiated by the component. If a component's provides-port is connected to two or more uses-port from other components, there is no prescribed scheduling behavior for the order in which the external invocations are served. It is up to the component implementation to determine this.

2.3.2.2 Uses-Port

A **uses-port** can be viewed as a connection point on the surface of the component where the framework can attach (connect) references to provides-ports provided by other components or by the framework. Viewed from the inside of the component, a uses-port is an object that implements the service the component needs to use. The component makes calls on a uses-port reference to *use* the *provided* capabilities. A component may have zero or more uses-ports.

Depending on the framework, one or more provides-ports may be connected to a single uses-port and a provides-port may be *provided* to one or more uses-ports. In general, if a uses-port has a member function that returns a value, the number of providers will be restricted to be one. Furthermore in these cases, there must be a connected provider for the component to operate correctly.

2.3.2.3 PortInfo

The ports on a component are identified with a string name which is unique among all uses and provides-ports within that component instance. Each port implements a specific functional interface. A standard CCA interface, called PortInfo, describes the binding between the port's name and its interface type.

The official specification language of CCA is the Scientific Interface Definition Language (SIDL). The SIDL specification of PortInfo is given by :

```
package CCA{
interface PortInfo{
    string getType();
    string getName();
    void setType(in string type);
    void setName(in string name);
    string toString();
};
...
};
```

Figure 2.6: SIDL specification of PortInfo

PortInfo simply binds a text string name of a port with the fully qualified name of the service interface it implements. Port interfaces are specified in SIDL and then translated to the target component implementation language. For XCAT, Java or XML Schemas can be used as an IDL. The Open Framework CGT is used for generating language specific bindings. For example, a port of type “Echo” from the package *samples.idl.echo* that has a single member function *Print()* and takes a string and returns a string might be defined in SIDL as shown in Figure 2.7.

```
package samples.idl.echo;

public interface Echo {
    public String Print(String str);
}
```

Figure 2.7: Echo Interface

2.3.3 SIDL translation

To translate SIDL in target component implementation language, XCAT uses its own compiler which translates the Java specifications into C++ classes. XCAT uses XML Schema documents in order to perform the translation. Here is an example of such a mapping for string :

```
<mappingsInfo>
  <parameterTypeMapping>
    <uri>java.lang</uri>
    <localName>String</localName>
    <langType>string</langType>
  </parameterTypeMapping>
  ...
</mappingsInfo>
```

Figure 2.8: String mapping in XML

The Open Framework CGT generates seventeen files for each port. In addition, the user might need to pass objects as function parameters instead of primitive statements. These objects need bindings in the target language. The object declaration should include its member attributes. Five more files are generated per object. For example, if we change the specification of *String Print(String str)* to *String Print(TestObject testobject)*, the Open Framework CGT will generate the previous seventeen files plus five more files corresponding to the *TestObject* generated files. Those files are listed in Appendix A.

2.3.4 The Services object

When a component is instantiated by the framework, it is provided with an instance of an object that implements the components interface to the framework. This object is called the *Services* object and is used by the component to tell the framework about the ports that it provides and uses. This object is the component’s view of the framework. The SIDL interface to the Services object is given Figure 2.9.

```

package CCA{
    ...
    interface Services {
        array<PortInfo,1> getProvidesPorts();
        array<PortInfo,1> getUsesPorts();
        Port getPort(in string name);
        void registerUsesPort(in PortInfo name_and_type);
        void addProvidesPort(in ProvidesPort inPort, in PortInfo name);
        void releasePort(in string name);
        void addConnectionListener(in ConnectionListener l);
        ComponentID GetComponentID();
    }
};

```

Figure 2.9: Services Interface

To tell the framework that a specific provides-port is available, the component must call a function named *addProvidesPort(in ProvidesPort inPort, in PortInfo name)* with the reference to the port instance and a PortInfo object naming and describing its type.

Uses-ports are named and specified by the component, but manufactured by the framework. To tell the framework that the component plans to use a port of a particular name and type it calls the *registerUsesPort(in PortInfo name_and_type)* function.

The *getProvidesPorts()* and *getUsesPorts()* functions return an array of PortInfo objects that describe the current port objects the component supports.

To obtain the actual reference to the uses-port, the component calls *getPort(in string name)* with the port name. If that-uses port has not been previously registered with a call to *registerUsesPort(in PortInfo name_and_type)* the *getPort()* call throws an exception. If the uses-port requires a connected provides-port and the connection has not yet been made, the *getPort(in string name)* operation will suspend until the operation is complete or it will return with a null reference.

To use a provides-port the user calls *getPort(in string name)* and calls *releasePort(in string name)* upon completion.

Each time the framework makes a connection or disconnects between a uses-port and a provides-port, it will emit an event. The event is delivered to the components at the ends of the connection (user and provider), telling the component the name of the port that was connected and the type of port and name of the port it was connected to/disconnected from. This information is delivered as a CCAConnectionEvent which is described in Figure 2.10.

The connection event delivers the connection information as an XML record in a string available through the function *getSource()*.

The *GetComponentID()* function returns an object of type ComponentID defined in Figure 2.11.

```

class ConnectionEvent extends FrameworkSpecific_Event {
    void setSource(in string source);
    string getSource();
};

```

Figure 2.10: ConnectionEvent

```

interface ComponentID{
    array<PortInfo,1> getProvidesPorts();
    array<PortInfo,1> getUsesPorts();
};

```

Figure 2.11: ComponentID Interface

A ComponentID object only provides accessor methods and cannot be used to modify the component or access framework specific implementation details. This object, when serialized, is the handle that can be used by services which are implemented as CCA components to refer to other components. A component is allowed to access its own ComponentID which it may then pass to services or components that implement services.

2.3.5 The Component interface

Each CCA component must implement the Component interface (see Figure 2.12).

```

package CCA{
    interface Component {
        void setServices(in Services cc);
    }
};

```

Figure 2.12: Component Interface

The *setServices(in Services cc)* function is called by the framework after the component has been instantiated. The component saves the passed reference to the Services object as a private instance variable. Prior to returning from the *setServices(in Services cc)* call, the component must instantiate and add all its initial provides-ports and register all its initial uses-ports using this instance variable.

As an example using XCAT C++, we have two components which are TestPrinterComponent and TestGeneratorComponent. TestPrinterComponent prints what TestGeneratorComponent has generated. The following examples, Figure 2.13 and Figure 2.14, show how a uses-port (for the TestGeneratorComponent) and a provides-port (for the TestPrinterComponent) are registered. In both components, the string *echo.wsdl* refers to the port type.

The way the framework notifies a component to shut down is to call the *setServices()* function with a null value. When such an invocation of *setServices()* completes, it is safe for the framework to garbage collect the component.

```

void TestPrinterComponent::setServices(Services *services_) {
    if (services_ == NULL) {
        if (services != NULL) {
            delete services;
        }
    }
    services = services_;
    services->addProvidesPort(new EchoImpl(),
                             new PortInfoImpl("inputEchoPort",
                                                "echo.wsdl"
                                                )
                             );
}

```

Figure 2.13: TestPrinterComponent

```

void TestGeneratorComponent::setServices(Services *services_) {
    if (services_ == NULL) {
        if (services != NULL) {
            delete services;
            return;
        }
    }
    services = services_;
    services->registerUsesPort(new PortInfoImpl("outputEchoPort",
                                                "echo.wsdl"
                                                )
                              );
}

```

Figure 2.14: TestGeneratorComponent

2.3.6 Application Manager

To launch components, we need a creation service. This creation service exists in XCAT Java [15] but has not been implemented for the C++ version yet. So in order to start and connect C++ components the Java creation service has to be used. The creation and connection service can be invoked from a Jython [7] script. An example of such a script can be found in Appendix B.

Appendix

A 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 correspond 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

B 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"
```

C 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] OMG-IDL, visited 12-06-2001. http://www.omg.org/gettingstarted/omg_idl.htm.
- [5] XML, visited 12-06-2001. <http://www.w3.org/XML/>.
- [6] XML Schema, visited 12-06-2001. <http://www.w3.org/XML/Schema>.
- [7] Jython, visited 9-10-2001. <http://www.jython.org>.
- [8] 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*.
- [9] Grid Computing Laboratory at the University of California San Diego. AppLeS, visited 01-03-2002. <http://apples.ucsd.edu/>.
- [10] 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*.
- [11] Randall Bramley, Dennis Gannon, Madhusudhan Govindaraju, and Aleksander Slominski. XSOAP, April 2001.
- [12] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. WSDL, visited 12-06-2001. <http://www.w3.org/TR/wsdl>.
- [13] Holly Dail. Holly Dail Masters Thesis, 2002.
- [14] 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/>.
- [15] Extreme Computing Lab. XCAT Java, visited 01-03-2001. <http://www.extreme.indiana.edu/xcat/>.
- [16] Extreme Computing Lab. XCAT, visited 01-03-2002. <http://www.extreme.indiana.edu/xcat/>.

- [17] Ken Kennedy, Mark Mazina, John Mellor-Crummey, Ruth Aydt, Celso Mendes, Holly Dail, and Otto Sievert. GrADSoft and its Application Manager: An Execution Mechanism for Grid Applications, October 04 2001.
- [18] Extreme Computing Lab. CCAT, visited 01-03-2002. <http://www.extreme.indiana.edu/ccat>.
- [19] Mark Mazina, Otto Sievert, Holly Dail, Graziano Obertelli, and John Mellor-Crummey. GrADSoft : A Program-level approach to using the Grid, February 24 2001.
- [20] Microsoft. DCOM, visited 01-03-2002. <http://www.microsoft.com/com/tech/DCOM.asp>.
- [21] AT&T Research. GraphViz, visited 10-20-2001. <http://www.graphviz.org/>.
- [22] Dimitri van Heesch. Doxygen, visited 10-20-2001. <http://www.doxygen.org/>.
- [23] Juan Villacis, Madhusudhan Govindaraju, David Stern, Andrew Whitaker, Fabian Breg, Prafulla Deuskar, Benjamin Temko, Dennis Gannon, and Randall Bramley. CAT: A high performance, distributed component architecture toolkit for the grid. In *Proceedings of Eighth IEEE International Symposium on High Performance Distributed Computing Conference, Redondo Beach, California*, August 3-6 1999.