Symphony – A Java-based Composition and Manipulation Framework for Computational Grids

Markus Lorch, Dennis Kafura
Department of Computer Science
Virginia Polytechnic Institute and State University
Contact e-mail: mlorch@vt.edu

Abstract

We introduce the Symphony framework, a software abstraction layer that can sit on top of grid systems. Symphony provides a unified API for grid application developers and offers a graphical user interface for rapid collaborative development and deployment of grid applications and problem solving environments through compositional modelling following the data-flow paradigm. Symphony meta-programs and program components can be distributed, reused and modified. Together with Symphony a new security model is developed that extends existing security architectures to allow for collaboration of grid developers and users in permanent as well as ad-hoc working groups.

1 Introduction

Evolving meta-computing systems like Globus [1], Legion [2] and UNICORE [3] provide for the creation of computational grids that can incorporate widely distributed resources. The services these new technologies provide are mainly accessible to the user by command-line utilities or through application-specific user interfaces of grid applications. Such interfaces differ in usage and appearance depending on the problem domain and the underlying grid architecture. For computational grids to become versatile and easy to use a unified abstraction layer is needed.

The description of a sequence of operations to be performed on a computational grid can be viewed as a meta-program. The term “meta” is used because the individual operations (e.g. execute a program, move a file) are of a higher order when compared to the operations in normal programming. A meta-program can be represented by a directed graph whose nodes denote executable programs and data resources and whose arcs represent data flow connections. Such a graph is shown in Figure 1.

Symphony is a Java-based composition and manipulation framework for computational grids. Meta-programs spanning multiple resources on possibly differing grid architectures and grid software can be composed, manipulated and executed using Symphony components. The Symphony framework abstracts grid architectures and their middleware. It can be used either as a component based graphical user interface or as an application programming interface (API) to standardize access to underlying grid services.

The next section gives an overview of the Symphony framework, how it has been implemented and what services it offers. Section 3 looks at the security architecture required to support collaboration of users in static and ad-hoc working groups. In Section 4 we present the Site-Specific Systems Simulator for Wireless Communications (S^4W). S^4W is a problem solving environment (PSE) for wireless system design. We used Symphony as a high level cohesive layer to tie together S^4W components and provide a GUI to the PSE. Section 5 refers to related work and Section 6 concludes with a summary and an outlook to future work.

2 The Symphony framework

Symphony is a component-based framework for composing, saving, sharing, and executing meta-programs [4]. The framework has two principle elements: a composition and control environment in which a meta-program is constructed and a back-end execution environment in which the described computation is performed. The composition and control environment can be a graphical user interface within which components are instantiated and customized to describe specific programs and files. These individual components can then be connected by links denoting dataflow relationships. Individual components and complete meta-programs can be saved, restored for later use and shared with other

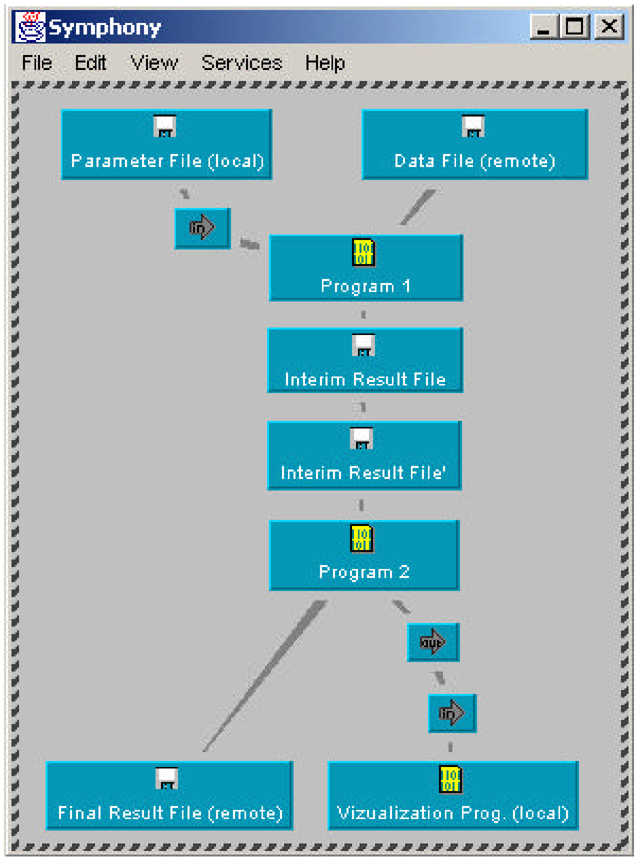


Figure 1: An example meta-program in Symphony

users. During execution, the components initiate and monitor the operations performed in the back-end execution environment. The operations are performed so as to respect the defined dataflow relationships. The back-end execution environment can use a proprietary server to access remote resources or use Globus toolkit [1] services. Symphony's architecture allows for the support of other back-end environments as well.

The Symphony Framework is based on Sun's JavaBeans™ component architecture [5]. The composition and control environment supplied with Symphony is a modified version of Sun's BeanBox. The BeanBox interface is shown in Figures 1 and 4; it allows generic Symphony components to be selected from a palette (shown at the left in Figure 4) and dropped into the BeanBox workspace. The properties of components in the workspace can be changed through customization. A Symphony bean can be customized to contain information about a resource for which the bean is a surrogate. Connections between Symphony beans allow the beans to synchronize the execution of the remote resources they represent.

Symphony is based on a small number of core beans, a protocol for the beans to communicate and synchronize, and a number of special purpose helper beans. The core beans are useful across all applications. Additional beans can be developed and added on an as-needed basis to provide for more specialized requirements. The Symphony framework provides base classes for such extensions.

The core beans in Symphony are:

- **The Program Bean**, represents a program to be executed at a local or remote resource.
- **The File Bean**, represents a local or remote file that may be pre-existing or may be created dynamically during execution of the meta-program.
- **The Standard Stream Beans**, provide for redirection of the standard-in, standard-out and standard-error streams.

An example of a simple meta-program is depicted in Figure 1. The meta-program consists of a remote program (Program 1) which takes a parameter stream and a data file as input and then generates an intermediate result file. This file is moved to a different resource where it is used by another program (Program 2) to generate the final result which is permanently stored in a file. A summary of the computation results is sent to the standard output which is redirected to the standard input of a local visualization bean (a helper bean). A more complex example of a Symphony meta-program is provided in Figure 4.

Remote programs and files can be accessed using a number of different protocols and services. For job submission and control Globus GRAM services as well as Symphony's proprietary services based on Java RMI [6] are currently used. Files can be accessed through HTTP, HTTPS, FTP, GSIFTP and local services. Whenever possible Symphony uses third-party data transfer mechanisms for efficient data routing. The Globus Commodity Toolkit for Java (CoG) [7] provides access to Globus specific services. Other protocols and services can be added easily by providing adapter classes.

When customizing the beans a resource browser elicits resource specific data from grid information services (e.g., the Globus MDS) and XML based resource configuration files to aid in the manual selection of resources. A programming interface provides for the use of resource brokers as described in [9], [10] and [11] to make dynamic resource decisions.

In Symphony a remote program component is customized by supplying the following information to the Symphony program bean either through its customizer GUI (Figure 2) or by calling the bean's customization functions from a controlling program:

- a URL for the computational resource used
- a URL for the software resource used
- a job specification containing various parameters
- authentication and authorization information

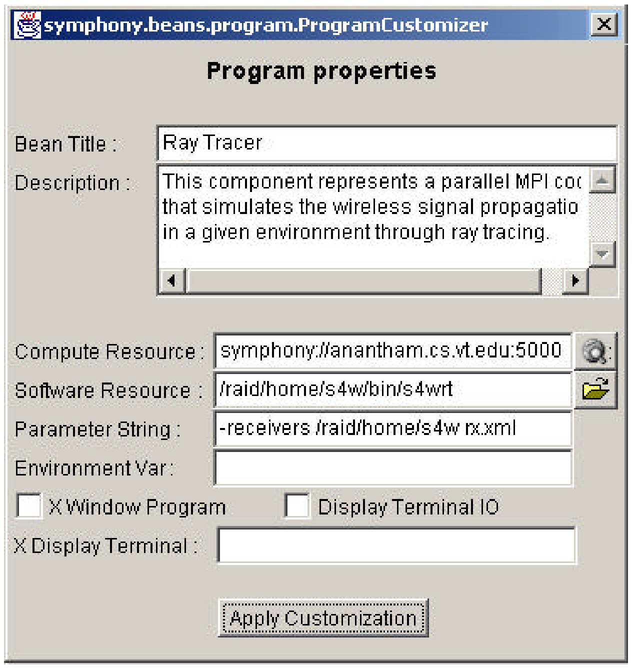
The first entity, the computational resource URL, is a standard resource locator that contains the protocol to be used to access the remote resource, the host name of the remote resource, if necessary the port number where the resource proxy can be reached and also the service that is desired at the resource. A sample URL for a Globus resource allocation manager (a.k.a. a Globus gatekeeper) is: "gram://gatekeeper.cs.vt.edu:2119/jobmanager". This syntax satisfies the requirements of many architectures. If the definition of a specific compute resource is not desired or necessary, due to the use of a resource broker, the URL contains a subset of the parameters, such as the protocol to be used, or the URL points to the resource broker engine.

The second entity is a URL that points to the software component to be used. In most cases this is just a location in the file system of the selected computational resource. If the executable has to be staged from another location separate from to the computational resource, then a full URL is used, e.g., "file:///home/mlorch/apps/myapp" for an executable located on the users workstation.

The third entity, the job description, may contain a set of name – value pairs which define the software resource that is to be executed and the environment requirements. This includes the values of environment variables, the working directory and location of the program if it has to be staged, as well as memory, CPU, architectural and software requirements. The job description also specifies whether or not the program's output streams should be displayed or discarded, if they are not redirected.

The fourth entity contains authentication and authorization information required to access the specified remote compute resource. It is not limited to one specific format. Currently support for X.509 certificates for authentication is provided and authorization is performed at the back end. Future extensions as described in Section 3 will also provide for authorization components.

Two operations can be performed on a constructed meta-program: verify and execute. Each operation can be initiated at any one program bean. During the verify operation the bean checks if the resource it represents is available and proper authorization exists to utilize the remote resource. A program bean uses its resource locations and job specification to dynamically generate a target-system dependent job specification, e.g., a Globus RSL string. The bean may, depending on the underlying architecture, allocate the remote resource to perform this verification. After successful verification of one bean's properties the verify signal is propagated to the bean's predecessors and successors, which in turn will propagate a verify signal after successful verification of their own properties. The execute operation applied to any selected program bean starts the execution of the entire meta-program. The execution operation is similar to the verify operation. The component on which the execution is initiated by the user informs its predecessors, if any, about its intent to start execution of the remote component it represents. If a component does not rely on any previous components to finish execution or data transfer it will start the execution of the remote component immediately. If a component in turn relies on other components to finish it will suspend itself until it receives clearance to proceed from these components.


symphony.beans.program.ProgramCustomizer

Program properties

Bean Title : Ray Tracer

Description : This component represents a parallel MPI coc that simulates the wireless signal propagatio in a given environment through ray tracing.

Compute Resource : symphony://anantham.cs.vt.edu:5000

Software Resource : /raid/home/s4w/bin/s4wrt

Parameter String : -receivers /raid/home/s4w rx.xml

Environment Var :

X Window Program

Display Terminal IO

X Display Terminal :

Figure 2: The program bean customization GUI

The user is notified by Symphony of the progress of the validate and execute operations through different colors of the components. Possible component states are: created, verified, running, completed and aborted. A fully configurable logging mechanism provides for detailed information on the course of a program execution.

The standardized architecture of JavaBeans provides for a large range of possible containers and operating systems that Symphony can work with. One such bean container is Sieve [8], which adds the ability for several users to remotely collaborate on the same meta-program. The Sieve container is a Java-applet which allows for simple access to the Symphony framework and thus to grid resources from web browsers. The small size and computational requirements of the Symphony beans and the portability of Java code enable Symphony to run on low-end workstations with a large variety of operating systems. Furthermore, the Symphony component framework is easily extended using a standardized, portable interface.

Grid applications can be developed independent of the underlying grid architecture by using Symphony beans as a grid application programming interface (API). Such an application can use the Symphony beans to access underlying grid services while providing a specialized user interface. The advantage of using Symphony as an API for grid applications is not only the independence from the deployed grid architecture but also the flow control features that Symphony provides. An application program can simply query the user for program parameters, instantiate and customize the corresponding Symphony beans and let Symphony take care of controlling and managing program execution and required data transfers.

The Symphony GUI can be used to construct and save a meta-program which can later be executed using the programming API. If little or no user interaction is required during program execution a simple application program can be used to load the saved meta-program and execute it without a graphical user interface. Information on the status of the program can be acquired by attaching a graphical component to visualize the program state, by using notification mechanisms such as system messages, or monitoring created log entries.

Symphony meta-program representations can be stored, shared and retrieved at a later time for execution or modification. Components that have been customized may be distributed to other users so that they can use them in other meta-programs with little or no additional modifications. Symphony uses inherited means for object serialization to achieve this. These persistence mechanisms are necessary to support the two groups of users in grid environments: Group One are those users who model meta-programs by combining single software and compute resources together to solve a complex problem. Group Two contains users who run such predefined meta-programs, alter the input data and parameters, and evaluate the created results. Symphony allows the two groups to easily trade meta-programs.

3 The Symphony security architecture

The Symphony framework requires a security architecture that provides finer grained and more flexible control of rights than is currently available on computational grids. The requirement for finer grained control arises from Symphony's ability to exchange meta-program components and whole meta-programs between users as part of their collaborative work. The creator of components or meta-programs can equip them with a minimal set of access rights to perform their computation when executed by another user. These rights are implicitly transferred to the user who receives and executes the components. Revocation of rights contained in the components becomes, of course, problematic and has to be dealt with through revocation lists at the resources. Greater flexibility is required because a given user may receive and combine components from many different sources. Current grid security infrastructures do not support these requirements.

In its current state, Symphony's security mechanism is based on the Grid Security Infrastructure (GSI) [12]. Every component in a Symphony meta-program can be equipped with a GSI proxy certificate (PC) that delegates the creators rights to the component. This enables Symphony components to access the resources that they are a surrogate for by impersonating the user that customized the component. If no designated GSI proxy certificate has been assigned to a component a default proxy is used which impersonates the user who is executing the meta-program.

This approach allows us to combine rights from collaborating users into one meta-program while still being layered on top of the security mechanisms put in place by the underlying grid middleware and grid resources. However, our approach also implies security issues that need to be solved before collaboration between non-trusting users can be supported securely. In their current implementation, the GSI proxy certificates cannot be limited except for their validity period and thus delegate all rights associated with the issuing entity, violating the least privilege principle [13]. This has been recognized by the GSI team and improvements are being developed [14, 15]. Future versions of the GSI will support the instantiation of very limited proxy certificates that can be tailored to only provide the rights necessary for a specific task through the inclusion of usage policies. With these mechanisms in place a Symphony component can be outfitted with a minimal set of necessary rights. If such a proxy certificate is extracted from a Symphony component by a malicious user it is of little use when applied in ways other than intended. Another drawback of the use of GSI proxies is that only one proxy credential can be submitted to access a resource. For accounting and security reasons a request should be accompanied by credentials that identify the current user as well as credentials that authorize the access.

Future collaboration in computational grids will partially consist of ad-hoc working groups that form through direct interaction between users from separate administrative domains. The administrative overhead of user creation and group membership association on the resource level prevents such collaboration. Mechanisms that allow for the delegation of rights from one user to another and the combination of rights from several sources are needed. The use of proxy certificates and similar approaches is a preliminary step that allows the incorporation of existing security infrastructures.

Distributed access mechanisms that allow resource owners to delegate rights to users without the intervention of a centralized administrative entity are a promising approach. The Akenti project [16] introduces two security components: use-conditions and attribute-certificates (AC). The owner of a resource, i.e. the owner of a computational, software, or data resource, generates use-conditions that are signed documents stating the ways in which the resource can be used. They also generate

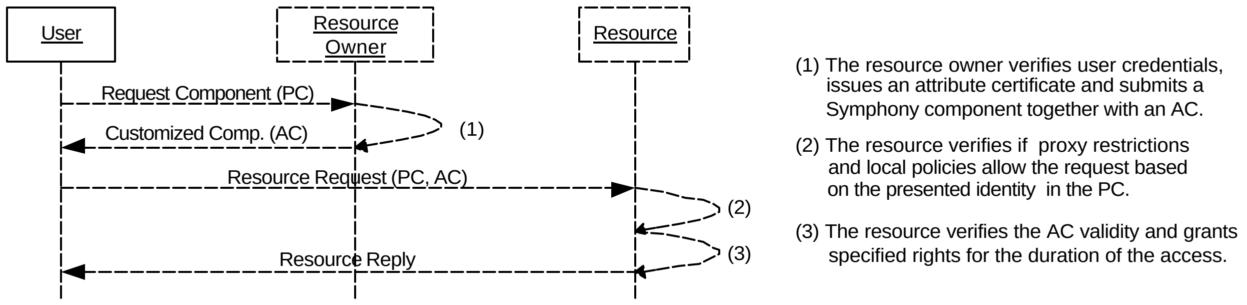

User
Resource Owner
Resource
Request Component (PC)
Customized Comp. (AC)
(1)
Resource Request (PC, AC)
(2)
(3)
Resource Reply
(1) The resource owner verifies user credentials, issues an attribute certificate and submits a Symphony component together with an AC.
(2) The resource verifies if proxy restrictions and local policies allow the request based on the presented identity in the PC.
(3) The resource verifies the AC validity and grants specified rights for the duration of the access.

Figure 3: Envisioned delegation scenario using Symphony components

attribute certificates that assign capabilities to other users. The users can then present their attribute certificates together with their identity certificates to resources which then make authorization decisions based on the presented certificates and their stored use-conditions.

The combination of the GSI with authorization elements from Akenti appears to be a good choice to provide delegation among users in Symphony. In such a scheme attribute certificates (AC) can be used to delegate specific access rights to a distinct user. GSI proxy certificates (PC) in turn impersonate the user and provide for authentication, accounting and coarse grained authorization. Figure 3 illustrates this mechanism: a user who owns a resource can create a Symphony surrogate for his resource and distribute this component. When another user requests to use the resource, the resource owner will issue an attribute certificate to that user. The AC certifies that a set of rights has been assigned to the specific user. The requesting user can then equip the Symphony component with this attribute certificate as well as with his identity proxy to use the resource. In our scheme revocation of issued attribute certificates is possible by simply adding specific attribute certificates to a revocation list at the resources.

The need for fine grained delegation of rights by the grid middleware and the enforcement of the resulting access policy by the grid resources poses another problem. Grid middleware systems that implement their own authorization mechanisms (e.g., Legion) can enforce fine grained policies. The GSI in contrast does not provide the granularity needed to support such enforcement as it depends on the underlying operating system for fine-grained authorization of resource access. The GSI and UNICORE provide for authentication of global users and their mapping to local user accounts at grid resources and are constrained by the expressiveness limitations of the resource operating systems. Extensions to the authorization mechanisms of the resource operating systems need to be in place to enforce complex usage policies stated in restricted proxy certificates and to grant rights that are conveyed through attribute certificates.

Most often grid resources run flavours of Unix which only provide static and coarse grained authorization mechanisms based on user and group IDs. User level sandboxes [17] can provide the necessary control over processes but place considerable overhead on the compute intensive applications often found in grid environments. Sandboxes are very much dependent on the system architecture and operating system and thus limit portability. A sandbox approach also would still require administrator intervention to modify group membership for collaboration among users of the same domain and lack mechanisms that allow for simple intra-domain collaboration. A more promising approach is the use of security extensions as defined in POSIX.1E [18], notably file system access control lists (FACLs). These operating system extensions are available for a fair number of leading operating systems used on grid resources such as Linux [19], Solaris [20], IRIX [21] and FreeBSD [22]. FACLs can be modified dynamically and provide the necessary means to only assign a set of minimal rights for many types of resource access.

Another approach to circumvent the expressiveness limitation is to perform authorization in the application code that serves a user request as done in the Community Authorization Service CAS [32]. CAS implemented the Generic Authorization and Access-Control API [31] which enables an application (e.g. a ftp server) to make fine-grained authorization decisions for resource access based on the applicable policy. This approach has the disadvantage that legacy services can no longer be used securely.

We currently prototype a security architecture that uses attribute and proxy certificates as explained above together with FACLs to support secure collaboration in ad-hoc working groups through the Symphony framework which will be described in more detail in a future paper.

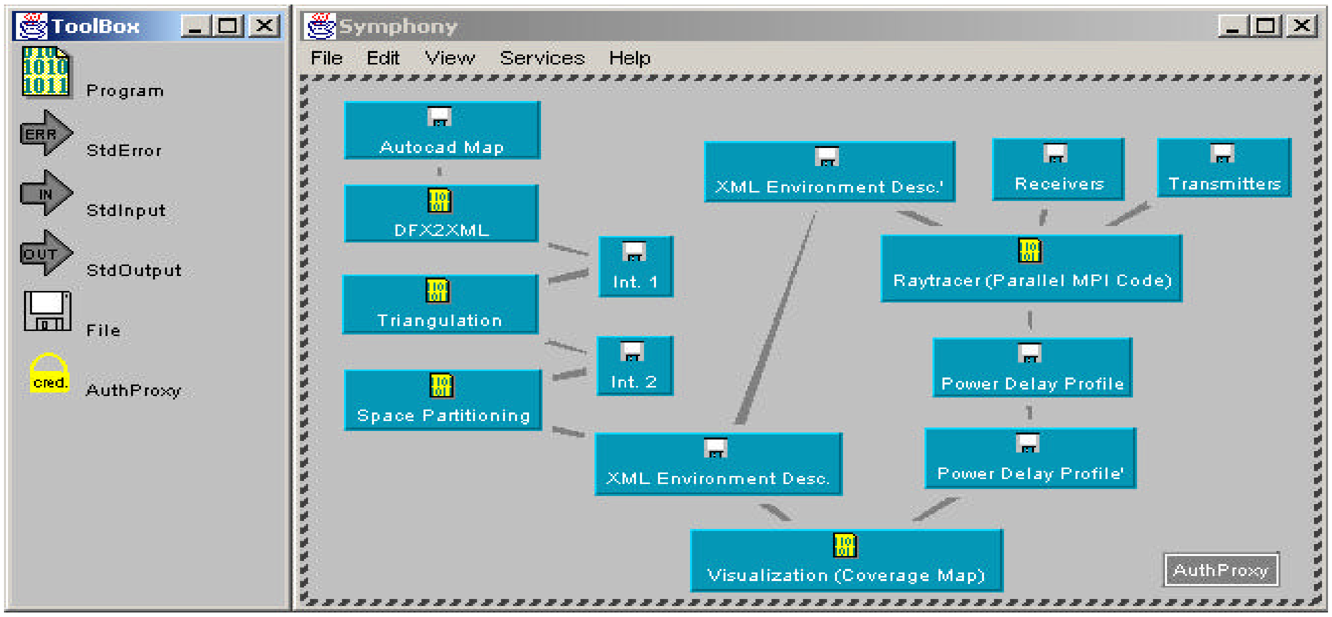


Figure 4: Application of Symphony to a Wireless System Simulation

4 Application of Symphony to a PSE for wireless system design

The Site-Specific Systems Simulator for Wireless Communications (S^4W) [23] is a simulator for wireless system design developed at Virginia Tech. S^4W uses a ray-tracing approach to model wireless systems and to predict coverage and bit-error rates. The system takes parameters such as environment obstacles, building materials, transmitter and receiver number and location, and movement of mobile equipment into account. S^4W can be used to optimize the locations of base stations in a cellular phone or wireless network system.

The Symphony framework allows for rapid collaborative development and modification of S^4W simulation meta-programs. Figure 4 shows a Symphony meta-program composed of S^4W components to optimize the location of base stations in a wireless network.

The illustrated example uses an environment map as input which it converts into a XML representation of polygons. A triangulation and space partitioning step transform this representation into an XML environment description suitable for the ray-tracer component. These pre-processing programs are typically run on a regular Unix workstation. The resulting XML file is then copied to a Linux cluster on which the ray-tracer component, a parallel MPI application, is executed. The ray-tracer takes transmitter and receiver locations as well as the environment description as input and saves a power-delay-profile as a result of the ray tracing. The file containing the results it then moved to a workstation with a visualization component that shows the predicted coverage.

5 Related Work

The Symphony framework can be viewed as layered on top of grid systems like Globus [1], Legion [2] and UNICORE [3]. Symphony's architecture allows for the incorporation of higher layer services offered for example by resource brokers such as Nimrod [10], AppLes [9] and EZ-Grid [11] as well as for the use of resource services directly when a grid infrastructure is not present. The following projects also allow the user to model grid applications using the workflow paradigm.

TENT [24] is a component-based framework that allows for the creation of complex grid applications using the graphical programming paradigm. TENT is mainly targeted at large scale computationally intensive simulations in the field of aircraft and turbine design. TENT is also based on the JavaBeans component architecture but uses CORBA [25] as its communication middleware. It provides for computational steering and can incorporate modules for the visualization of results. TENT is a vertically integrated solution, however, current work is aimed at using the Java CoG Kit for Globus to layer TENT on top of the Globus middleware.

WebFlow [26] is also a Java-based framework to develop high-performance meta-applications that run on distributed resources but is no longer an active project. A three tier architecture employed Globus services at the lower end. WebFlow object servers provided component interoperability using CORBA in the middle tier and at the top a WebFlow front-end together with powerful visual authoring tools were used to create and control meta applications and to provide for collaboration among users.

The Task Mapping Editor (TME) [27] is an integrated solution to provide access to remote supercomputers through its own server daemon. It provides a GUI in the form of a Java applet. TME represents remote resources as icons on a canvas. The icons can be interconnected to form a data-flow diagram following the same visual programming principles as used in Symphony, TENT and WebFlow. Execution and control of remote components, staging of data and the transport of data from one component to the next is handled by TME proprietary functions. TME is not yet released to the public. Current plans are to extend the system to use secure connections and to make improvements to the way data transfers are handled.

UNICORE [3] and its current successor UNICORE Plus aim at developing a grid infrastructure that facilitates access to distributed supercomputers for engineers and scientists through a graphical user interface. UNICORE lets the user define a task as a set of interdependent programs and data transfers to be executed on the remote resources and visualizes this in a data- and control-flow-graph. These task descriptions can be stored, modified and reused. Changes to legacy program components are not necessary. The user can view the state of the meta-program and download output data through a unified interface. Public key certificates are used to authenticate users at remote resources via credential mapping.

The Directed Acyclic Graph (DAG) Manager DAGMAN is a scheduler component of the Condor project [30] that allows specification of job interdependencies by creating a DAG and manages the execution of the jobs with respect to those interdependencies. DAGMAN is a stand-alone command line utility; an API for interoperability with other components is planned.

CCAT [28] and its successor XCAT are projects at Indiana University based on the Common Component Architecture (CCA) [29]. A sophisticated service architecture has been developed that provides for the creation and discovery of instantiated components and component descriptions as well as for event notification and connection mechanisms. XCAT components use Globus services through the Globus Java COG kit for security and task creation mechanisms and employs Java RMI for communication between components.

The main difference between the approaches taken in the Symphony project and the TENT, WebFlow, CCAT and XCAT architectures is the focus on legacy applications in Symphony. For TENT existing applications have to be modified to use the TENT API in order to communicate with other components. In the case of WebFlow a great number of the problem solving modules were specially developed Java components. CCAT/XCAT are frameworks for CCA components.

Symphony in contrast is a framework for combining arbitrary codes to meta-programs. Existing programs can be tied into a meta-program without changes to the code. Symphony components can be implemented in the language and on the architecture suited best for the application and legacy codes do not have to be ported.

Another significant difference is the size, complexity and extensibility of the solutions – TENT and WebFlow rely on their own server components being in place in the distributed environment. CCAT and XCAT are powerful but rather large and complex frameworks. UNICORE and TME are vertically integrated solutions that require a specific infrastructure. Symphony, however, provides a small-footprint framework that can be used on top of a computational grid without the need to install remote components or a service infrastructure. Symphony can also access resources that lack a grid middleware installation through its proprietary lean back-end. Local applications like visualization tools can be incorporated into a Symphony meta-program without modifications. Symphony should be seen as a lightweight workflow abstraction and user interface framework for computational grids that provides a valuable alternative to the aforementioned projects.

6 Concluding remarks and future work

Symphony is an abstraction layer for existing grid middleware and operating systems. It provides support for collaborative rapid development of meta-programs through compositional modeling and can serve as a unified API to grid environments allowing the system independent application development. The open architecture of the Symphony framework based on JavaBeans ensures the portability, interoperability and extensibility required by high level abstraction software for computational grids.

Future aims are additional support for other grid middleware software such as Legion and UNICORE. Interoperability with the DAGMAN execution manager and integration of Symphony components into the XCAT framework will be investigated in more detail.

Our immediate goals are to develop and implement a new security architecture that supports collaborative work in ad-hoc as well as more permanent user groups. More Information on the Symphony project can be found at http://symphony.cs.vt.edu.

References

[1] I. Foster, C. Kesselman, ”Globus: A Toolkit-Based Grid Architecture” in “The Grid, Blueprint for a Future Computing Infrastructure”, Foster, I. and Kesselman, C. Editors, Morgan Kaufmann, 1999, pp. 259-278

[2] A. Grimshaw et al. “Legion: An Operating System for Wide-Area Computing” IEEE Computer, 32:5, May 1999: pp. 29-37.

[3] M. Romberg "UNICORE: Beyond Web-based Job-Submission" Proceedings of the 42nd Cray User Group Conference, May 22-26,2000, Noordwijk

[4] Ashish Shah, "Symphony: A Java-based Composition and Manipulation Framework for Distributed Legacy Resources", Master Thesis in Computer Science, Virginia Polytechnic Institute and State University, 1998

[5] Sun Microsystems, "The JavaBeans^{TM} Component Architecture", http://java.sun.com/products/javabeans/, 2001-11-05

[6] Sun Microsystems, "Java Remote Method Invocation", http://java.sun.com/products/jdk/rmi/, 2001-11-05

[7] G. von Laszewski et al, “CoG Kits: A Bridge between Commodity Distributed Computing and High-Performance Grids”, ACM 2000 Java Grande Conference, 2000

[8] P. L. Isenhour, et al, "Sieve: A Java-based collaborative visualization environment", Proceedings of the IEEE Visualization'97, Phoenix, AZ, Oct. 1997, pp. 13-16

[9] F. Berman and R. Wolski ”The AppLeS Project: A Status Report “, Proceedings of the 8th NEC Research Symposium, Berlin, Germany, May 1997.

[10] R. Buyya et al, “Nimrod/G: Resource Management and Scheduling System in a Global Computational Grid”, The 4th Int. HPC Asia Conference 2000, Beijing, China. IEEE Computer Society Press, USA, 2000

[11] B. Chapman et al, "EZ-Grid Resource Brokerage System", http://www.cs.uh.edu/~ezgrid/, 2001-11-05

[12] I. Foster et al, "A Security Architecture for Computational Grids", Fifth ACM Conference on Computers and Communications Security, Nov. 1998

[13] J. R. Salzer and M. D. Schroeder, "The Protection of Information in Computer Systems" Proceedings of the IEEE, Sept. 1975

[14] B. Sundaram et.al, "Policy Specification and Restricted Delegation in Globus Proxies", Research Gem, Supercomputing Conference 2000, Dallas, November 2000

[15] K. Jackson et al. “TLS Delegation Protocol”, http://www.ietf.org/internet-drafts/draft-ietf-tls-delegation-01.txt, July 2001

[16] M. Thompson et. al, “Certificate-based Access Control for Widely Distributed Resources“, Proceedings of the Eighth Usenix Security Symposium, August 1999

[17] I. Goldberg et al. “A secure environment for untrusted helper applications: confining the wily hacker”, Proceedings of the 1996 USENIX Security Symposium

[18] IEEE standard portable operating system interface for computer environments**,** IEEE Std 1003.1-1988 , 1988

[19] LIDS Security Extentions for Linux, http://www.lids.org, 2001-11-05

[20] Sun’s Trusted Solaris 8 Operating Environment, http://www.sun.com/software/solaris/, 2001-11-04

[21] Trusted IRIX, http://www.sgi.com, 2001-11-05

[22] POSIX.1E: Mandatory Access Control Support for FreeBSD, http://www.watson.org/fbsd-hardening/posix1e/mac/, 2001-11-05

[23] A. Verstak et. al, “Lightweight Data Management for Compositional Modeling in Problem Solving Environments”, in Proceedings of the High Performance Computing Symposium, Advanced Simulation Technologies Conference, Apr. 2001, pp. 148-153

[24] T. Forkert, et al, “The Distributed Engineering Framework TENT**”**. In Proceedings of Vector and Parallel Processing - VECPAR 2000, LNCS 1981, 2000, pp. 38-46

[25] CORBA Specifications http://www.corba.org, 2001-11-05

[26] E. Akarsu, F. Fox, W. Furmanski and T. Haupt “WebFlow - High-Level Programming Environment and Visual Authoring Toolkit for High Performance Distributed Computing”, Proceedings of Supercomputing 1998

[27] The Task Mapping Editor http://ccse.koma.jaeri.go.jp/projects/Projects_data/data_093/projects_093.html, 2001-11-05

[28] R. Bramley et al. “A Component-Based Services Architecture for Building Distributed Applications”, Proc. of the 10th Int. HPDC Symp. Aug. 7-9, 2000, San Francisco

[29] “The Common Component Architecture Forum”, http://www.cca-forum.org/, 2002-01-10

[30] J. Basney. and M. Livny, "Deploying a High Throughput Computing Cluster", High Performance Cluster Computing, Vol. 1, Chapter 5, Prentice Hall PTR, 1999.

[31] T.V. Ryutov et al.“An Authorization Framework for Metacomputing Applications”, Cluster Computing Journal, Vol. 2 Nr. 2, 1999, pp. 15-175

[32] L. Pearlman et al., “A Community Authorization Service for Group Collaboration”, submitted, 2002 IEEE Workshop on Policies for Distributed Systems and Networks, http://www.globus.org/Security/CAS, 2001-01-10