Editor: Geoffrey Fox, gcf@indiana.edu

GRIDS OF GRIDS OF SIMPLE SERVICES

By Geoffrey Fox

IN PREVIOUS INSTALLMENTS, WE ADOPTED THE VIEW THAT GRIDS REPRESENT THE SYSTEM FORMED BY THE DISTRIBUTED COLLECTION OF ELECTRONIC CAPABILITIES MANAGED AND COORDINATED TO SUPPORT SOME SORT OF ENTERPRISE (A VIRTUAL organization). Sometimes, we use "grid" to describe just the technology used to build these electronic communities or organizations. We think of grid technology as the cyberinfrastructure (the US National Science Foundation) or e-infrastructure (European Union) that supports e-science, e-business, or, in fact, e-more-or-less-any-enterprise.

In this article, I describe how to build systems from service-oriented grids that let you build new grids by composing and adapting existing collections (libraries) of grids. I also suggest some best practices for deciding how to architect services and package systems.

There is no firm consensus on the best grid approach but most people would use Web services. There is a vigorous community debate on the "right" way to do this and whether Web services need enhancements to cope with a grid's large-scale, secure, managed distributed services. In particular, there is much discussion on appropriate representation of *service state* and its standardization. Service state refers to the way the service records its current definition; for example, in an online shopping service, what is in a shopping cart and whose cart it is.

The Web Service Resource Framework (WSRF; www.globus.org/wsrf) and the Web Service Grid Application Framework (WS-GAF; www.neresc.ac.uk/ws-gaf) are two important activities whose development and interaction will have important implications for Web services' detailed structure and the way state is specified. However, I'm talking here about aspects independent of these issues—namely, the right size for a service and how to package services and grids together.

Services

Often we consider grids as providing seamless access to a set of resources. I agree but also propose that the resulting grid architecture can consist of many small grids. This reflects the many different overlapping community types and resource collections that naturally form individual grids. Each individual grid can have a seamless elegant environment—this even could be a criterion for defining basic grids—but a composite grid would amalgamate multiple subgrids and provide a resultant heterogeneous environment. In other words, we don't want just a few grids but a large number composed, divided, and overlapped to support dynamic communities and requirements.

The service-oriented architecture (SOA) that grids use today differs subtly from earlier distributed systems built with Component Object Model (COM), Corba, and Java, and includes enhancements, especially in interoperability and scalability. Key Web-services features in today's grids include

- Architectures that choose, wherever possible, message-based—not method- or Remote Procedure Call (RPC)-based—capabilities linkage. This produces lightweight, loosely coupled services that can be distributed and replicated to achieve needed performance and functionality.
- Interfaces defined with XML-based SOAP and Web Services Description Language (WSDL) technologies, which support a wide set of implementations that trade off performance, ubiquity, and functionality.

Providing an accurate definition of loose message-based coupling is not easy. A traditional distributed object model produces components that typically exchange messages with an RPC or equivalent Java remote method invocation (RMI). These coupled messages correspond to the distributed version of a traditional method call and its return. Loose coupling for services corresponds to a messaging strategy where individual messages are not directly coupled in pairs, and, if needed, response messages are generated asynchronously from the original communication. The second key services feature—XML-based specifications of the service interfaces and their associated messages—is important for interoperability but less distinctive in its ar-

Copublished by the IEEE CS and the AIP 1521-9615/04/$20.00 © 2004 IEEE

chitectural implications; it roughly corresponds to a different specification language from Corba's Interface Definition Language (IDL) or Java's RMI.

Choose any software problem facing you today and imagine how it would look in a traditional approach of a decade or so ago. We would get a giant glob of software in some language, such as C++ or Fortran. The software problem would be divided into methods or subroutines and we would be browbeaten to build it in a modular fashion using libraries and well-defined interfaces. Today, we people from the past have given up using `GOTO` in Fortran and adopted better practices for specifying control structures. As technologies developed, we added new languages, such as Java, and better software engineering processes, which industry adopted more broadly than academia.

As I already implied, distributed object technology supported this paradigm implementation across multiple computers, with method or procedure calls implemented as paired messages. However, most software systems still consisted of large globs, each of which had multiple functionalities. You can find many very useful and important examples of this for Java at www.apache.org.

You can convert that code into services by specifying each of the interfaces in XML and providing a Web-service wrapper. This activity is important for jump-starting your services collection, but it is an interim step. For example, if you look at all the different Apache projects, you will find many related but different implementations of common subservices, such as security and user profiles. Building a system that combines several projects often requires an integrated approach to common services, which would be relatively easy to do if each subservice implementation were a separate grid service with well-defined message-based interfaces. However, with a traditional approach, a typical subservice such as security might have an external message-based interface but, unfortunately, also many internal methods linking the subservice to other parts of the software glob. Thus, subservices can't be extracted from the glob, so composing such traditional software systems, even if they run smoothly and efficiently with service interfaces, is very hard.

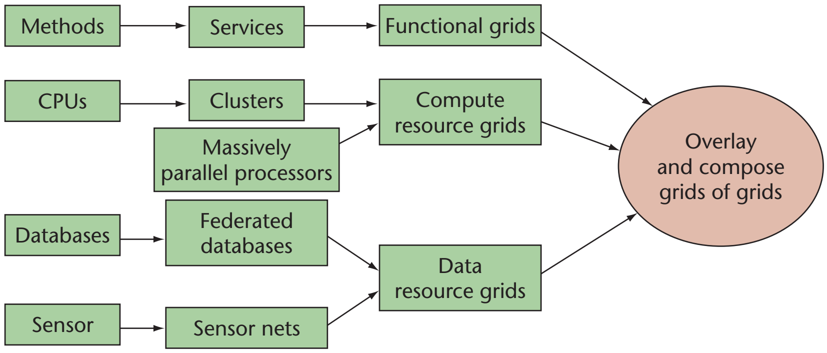

Figure 1. Composing functionality and resources in a grid of grids. This figure illustrates several different hierarchical packaging including those of traditional software engineering, CPU clusters, federated databases, and sensor nets. The grid-of-grids concept generalizes these ideas.

Taking all this into consideration, let's identify a strategy for defining services. Start by examining the different capabilities of your systems. Services are distributed components that have distinct functionality—especially functionality shared usefully among different uses.

Services must achieve acceptable performance when implemented with message-based interfaces and distributed platforms. In an earlier installment ("Making Scientific Applications as Web Services," vol. 6, no. 1, 2004, pp. 93–96), we discussed the inevitable latency differences between message- and method-based interactions; messages could experience hundreds of milliseconds in network latency down to a millisecond or so for communication between nearby services.

We should build services that are as small as possible given the performance implications from decomposition. Services are the package created by traditional programming models so languages apply. Rather than discussing this aspect, however, we'll look at a higher level, with services as the atomic unit whose management and packaging into grids needs to be explored. I use the term "simple service" in this article's title to refer to services constructed in this fashion to be as small as possible given inevitable performance and functionality constraints.

Packaging Services and Resources into Grids

In this article, grids represent a packaging and coupling approach that generalizes and distributes a familiar progression taken from the traditional software hierarchy: lines of code → methods (subroutines) → objects (programs) → packages (libraries). Figure 1 shows that we can consider grids this way, using a service or a resource as the basic building block. However, a given grid is not the last word; it can be a building block in a larger grid. Thus, I propose building systems as grids of grids, with single services or resources viewed as a special case of (small) grids.

In Figure 1, I chose to separately specify grids that correspond to resources (made up of data repositories, sensors, and CPUs) as well as those corresponding to functionalities (software services). This can be confusing because every grid resource is represented

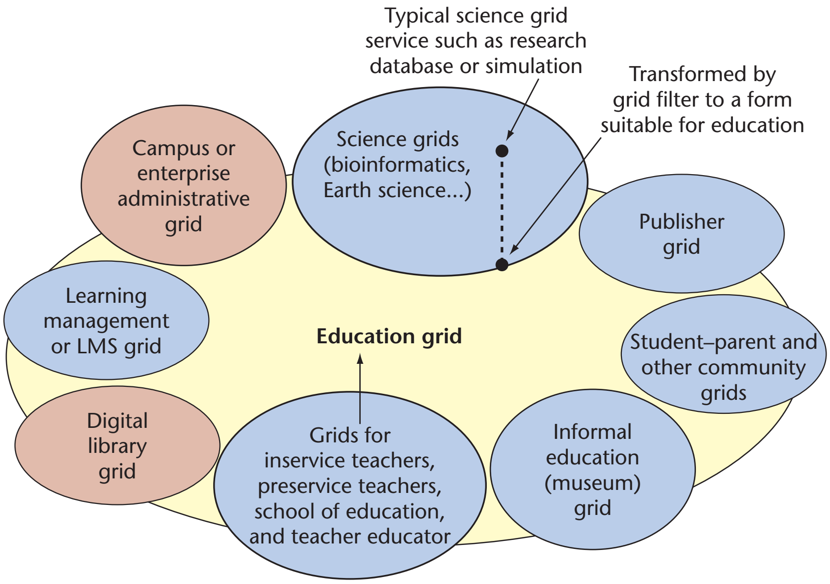

Figure 2. Science education as a grid of grids. One approach to building an education grid that exploits relevant resources and communities that we'd expect to be independently organized into grids.

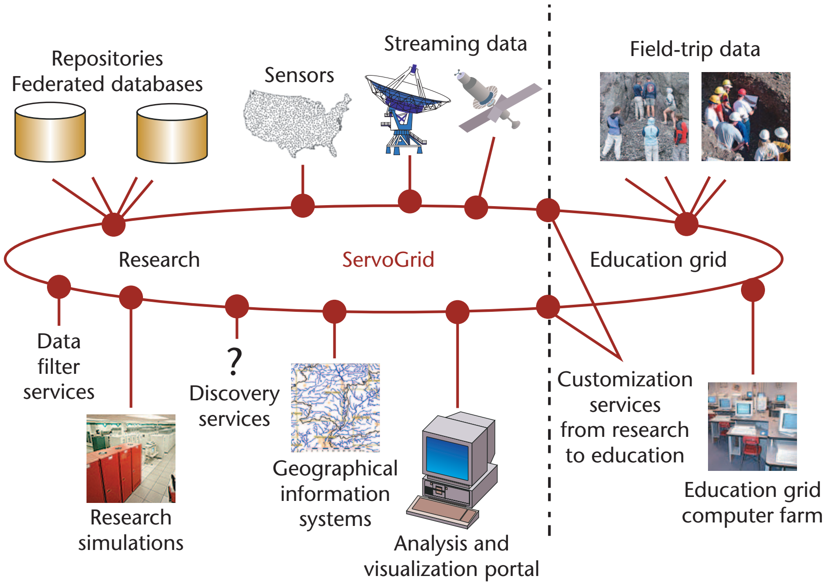

Figure 3. Geoscience research and education grids. ServoGrid is a science research grid built by a team led by the Jet Propulsion Laboratory and aimed at solid Earth research. Research activities are on the left and a geoscience education grid is on the right.

by a service. Thus, we could simplify all this and just talk about services.

This approach provides some unification of well-known concepts; for example, an individual grid service could correspond to a single database using the Open Grid Service Architecture–Data Access and Integration (OGSA–DAI) technology described in an earlier installment ("Integrating Computing and Information on Grids," vol. 5, no. 4, 2003, pp. 94–96). A federated database then would correspond to a database grid. As another example, an individual CPU could have a grid service interface; a cluster grid would correspond to a cluster of CPUs aggregating the individual CPU simple services.

Let's look at another example, this time from education: science grids in schools and universities. As Figure 2 shows, education involves many separate communities and capabilities that form independent electronic (virtual) organizations supported by their own grids. We create an education grid using a grid of grids by linking and adapting services in the component grids.

Traditional specialized educational services are organized as a learning-management grid. A digital library grid could organize and deliver knowledge; a campus grid offers digital registration services; teacher–educator grids link preservice (school of education) and inservice teaching grids complemented with museums via an informal educator grid. The learners, parents, and other education stakeholders all naturally form their own grids.

Finally, science education could be addressed in this framework by linking to research science grids such as that of ServoGrid (Figure 3). These component grids are interfaced and composed with transformation services to form a science education grid of grids. The research grid includes databases, field data, sensors, filters (to preprocess data from sensors and databases), geographical information system (GIS) services organized as their own grid, and discovery and simulation services. Figure 3 also shows the needed portals and user interfaces to the grid of grids. In a previous installment ("Grid Computing Environments," vol. 5, no. 2, 2003, pp. 68–72), we described grid portal architectures involving portlets that support construction of portals for composite grids from user interface components for individual grids and services.

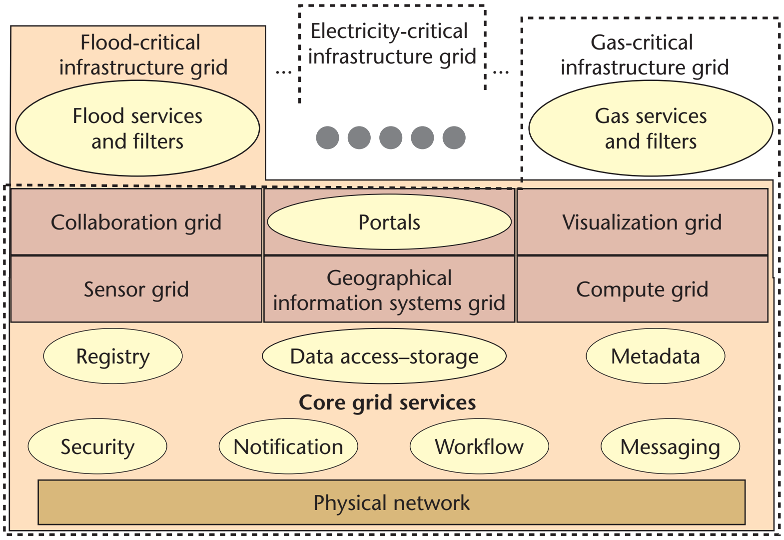

Figure 4. Critical infrastructure (CI) grids built in a composite fashion. A nation's CIs, such as water and electrical or natural gas power, can be organized hierarchically as a set of component grids. The latter include collaboration, visualization, sensor, GIS (geographical information system), and computing grids.

Figures 2 and 3 illustrate the key idea of using transformations or filters to adapt services in old component grids to the new education grid. This approach could take research simulation or database services and simplify them for use in education. The resulting education grid would consist of three service types: those unique to education, such as educational metacontent (lesson plans and objectives), online knowledge bases, and grading and homework services. These education-specific services are delivered by learning-management and digital library grids. The second category of service in an education grid of grids is illustrated by services like collaboration, which are essentially the same as those developed for other grids; the third category includes the transformed grid resources developed for research but transformed to directly support teaching and learning.

Thus, we first should build the simple services discussed earlier and then package them into atomic (building-block) grids covering core functionalities and services; geoscience, digital libraries, and learning-management systems are example atomic grids. Figure 3 shows a geoscience grid that uses a geographical information system (GIS) grid as a component. After defining the basic grids, we can build most operational grids by linking component basic grids and customizing them by adding services to filter or transform their services. Thus, our end result is a grid of grids.

Figure 4 shows another grid-of-grids example. It illustrates how we can build grids to support a national critical infrastructure (CI). The US Department of Homeland Security identified several such infrastructures: agriculture and food, water, health, industrial and defense, telecommunications, energy, transportation, banking and finance, chemical industry and hazardous materials, and postal and shipping.

In this case, the critical atomic grids include sensors, GIS, visualization, computing, and collaboration. Figure 4 also shows the core grid services we need, such as registries, databases, metadata, security, notification, workflow, and messaging. These core services and atomic grids are composed with infrastructure-specific services to form a particular CI grid of grids.

Figure 4 also shows how we can reuse atomic grids in all CI grids and illustrates important interoperability principles with which grids are built. These CI grids are, in turn, customized, composed, and overlaid with other grids (such as weather, census data, and so on) for different CI communities. Thus, we can generate grids aimed at public health, emergency response (command and control), or crisis management, infrastructure planning, education (schools), and training (managers and first responders). We can apply the grid-of-grids concept recursively and dynamically.

My approach builds grid systems hierarchically—using traditional software engineering to describe the structure of individual simple services—and aggregates them into atomic grids that perform core functionalities. Atomic grids are composed into higher-function grids of grids. Using transformation services in this integration of component grids distinguishes this packaging approach from that common to libraries.

Although a lot of research on workflow technology supports the composition of services (www.extreme.indiana.edu/groc/ggf10ww/index.html), it seems that no one has given much consideration to the capabilities of modern integrated development environments for traditional software models or to use them for the higher level of integration necessary in grids of grids. In fact, it is hard to support my suggestion to make services as small as possible given the poor support for managing them. I expect the ideas described here will receive increasing attention in the future with the growing importance of software engineering and its extension to services.

Geoffrey Fox is director of the Community Grids Lab at Indiana University. Contact him at gcf@indiana.edu; www.communitygrids.iu.edu/IC2.html.