

## Middleware

# Application Deployment on Catalactic Grid Middleware

Liviu Joita and Omer F. Rana • Cardiff University, UK

Pablo Chacín, Isaac Chao, Felix Freitag, and Leandro Navarro • Technical University of Catalonia, Spain

Oscar Ardaiz • Public University of Navarra, Spain

*An architecture based on a decentralized market view integrates Grid applications with Catalactic middleware. A prototype application showed the concept's feasibility, as well as the middleware's effectiveness in balancing query-request workload across multiple Grid services.*

Grid computing researchers have shown significant interest in using an economic paradigm for exchanging Grid resources and services.<sup>1</sup> With this approach, applications can use a market mechanism to schedule services access, thus giving them a fair, efficient way to share resources in high-demand periods. Most existing approaches rely on centralized brokers to coordinate resource access, and they're typically implemented over existing Grid middleware. We propose an alternative approach based on F.A. Hayek's Catallaxy mechanism.<sup>2</sup> Catallaxy's free-market, self-organizing coordination mechanisms adjust prices within the market based on the particular demands for a specific (scarce) resource.

In previous work, we simulated Catalactic Grid markets and compared them to centralized economic allocations.<sup>3</sup> Our satisfactory results encouraged us to investigate the feasibility of implementing Catalactic Grid middleware and integrating it with Grid applications. The Catallaxy approach is based on negotiation and price signaling between decentralized autonomous agents. (In our work, "agents" refers to autonomous service providers or users who can update or modify their services and determine how much service information is accessible to other agents.) Catallaxy lets applications inform individuals (agents) about other agents' possible knowledge and facilitates information exchange. This information leads to price generation that complies with the value each agent assigns to the information.<sup>3</sup> Catallaxy therefore enables the development of self-organizing, highly dynamic agents, thereby facilitating peer-to-peer (P2P) system behavior. Such an approach is particularly suited to open systems in which detailed knowledge about particular agents might be unknown a priori.

Here, we describe an architecture that integrates Grid applications into a market that supports the Catallaxy concept; we do this by integrating the application with a Catalactic middleware. We also describe a prototype based on a distributed database-query application that searches distributed catalogs for building products in the architecture-engineering-construction industry.<sup>4</sup>

## Overview: Grid markets

Developers can apply Catalactic free-market mechanisms to computational and data grids<sup>5</sup> that host the interrelated resource and service markets. In the Grid resource market, resource providers sell their computational, storage, bandwidth, or tool resources to resource buyers. The traded goods are physical resources that buyers use to execute their own applications. Because researchers expect Grid resource markets to include numerous sellers and buyers, such Catalactic free-market mechanisms will likely be required.

In the Grid service market, providers sell services to clients. Such services typically provide a particular application functionality, such as transcoding, query-execution, or molecule-docking services. Each service must therefore provide a self-contained functionality on the assumption that external services are fully independent. Service buyers use purchased services alongside their own existing services.

Service providers can buy resources on the Grid resource market to provide their services in the Grid service market. For example, a transcoding service provider might buy computational resources to execute its transcoding application for a particular client request. Although the resource and service markets depend on each other to some degree, each can operate somewhat autonomously by applying Catalactic mechanisms.

## **Catalactic middleware**

To implement Catalaxy in a real-world Grid application, we had to design a Catalactic middleware capable of dealing with different application scenarios. We developed this middleware to support Catalactic computational markets that offer high-level abstractions and mechanisms, including those that

- locate and manage resources,
- locate other trading agents,
- negotiate with agents, and
- adapt to changing conditions.

Given the potential variability in the application characteristics, our middleware should implement only general mechanisms so developers can plug in application-specific strategies and policies to suit specific scenarios.

## **Application scenarios**

Application scenarios for the Catalactic middleware are

- dynamic. Applications operate within a changing environment and thus must be adaptable.
- diverse. Requests have different priorities, and applications should respond accordingly.
- large. Because applications might have to coordinate numerous elements, locality is required if applications are to scale effectively.
- partially informed. It's impossible for any entity to have complete information about the system's state. Reasons for this include scale issues and communication latency, which requires locality (thereby leading to scalability problems).
- complex. Applications include numerous parameters and need learning mechanisms to self-adjust or adapt to changes, so optimal solutions aren't easily computable.
- evolutionary. Applications are open to changes that can't be accounted for in the initial setup; they can also learn and make decisions given limited information (about their neighbors, historic data, and so on).

## **Requirements**

The Catalactic middleware must meet several criteria to handle these application scenario characteristics. First, it should be scalable in highly dynamic environments. It should be able to address scenarios with thousands of nodes in an environment where nodes frequently enter and leave the network. The network configuration's dynamism implies that the middleware should maintain minimal system information (avoiding global topological information) and offer easy, efficient updates.

Second, the middleware must support heterogeneous environments. The environments' scale implies high heterogeneity among applications, the underlying platform, resources, providers' service

properties, and node availability (some will be quasi-permanent, but others will enter and leave).

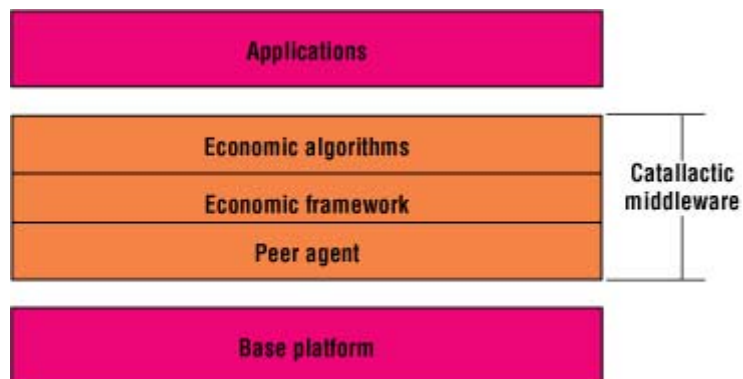
Third, the middleware should be compatible with different base platforms and thus define a set of generic APIs. Some users might need adaptors to translate this generic model to each platform's specific model. Such a translation mechanism could harm system performance if transformations are complex or frequent.

Fourth, the middleware must support component self-organization because network dynamics prevent a priori peer configuration and maintenance of centralized configuration files. Because peers must continuously discover network characteristics and adapt accordingly, the middleware must distribute important system functions—such as security, resource management, and topology management—that are traditionally reserved for specialized nodes.

Fifth, the middleware must support different implementation architectures because it might be deployed under different configurations. It's therefore important that individual components not make assumptions about a function's specific distribution among the middleware infrastructure's physical components.

## Architecture

A layered architecture provides a clear separation of concern among layers. This lets developers build a more adaptable system, because they can use pluggable rules and strategies to progressively specialize the upper layers into specific application domains. As figure 1 shows, our middleware supports five layers.



**Figure 1. The layered middleware architecture. Applications interact with the Grid market middleware to obtain the necessary services to fulfill a particular function. The base platform supports the applications that provide the Grid services' hosting environment.**

*Application layer.* This layer implements end-user applications such as collaboration tools and problem-solving environments. Applications rely on the base platform for communication, platform-level resource management, and other basic functions. Applications also have application-level resources, however, such as a collaboration tool's virtual meeting room or a scientific application's matrix resolution algorithm. The interaction model between the application layer and the Catalactic middleware is application and middleware dependent. Applications can manage their resources by directly interacting with the Catalactic middleware (becoming Catalactic-enabled) or by interacting transparently through the base platform they're built on.

*Economic algorithms layer.* This layer implements domain- and platform-independent economics algorithms for resource allocation. It also includes interacting agent services that act as sellers and buyers in service and resource markets, as well as extensions and specializations of the underlying framework's functionalities. It adapts these functionalities to the specific application layer network (ALN) and the existing resource-allocation policies.

*Economic framework layer.* This layer provides primitives that support Catalactic algorithm implementation. Among these algorithms are those for finding peer agents to negotiate with, starting negotiation, making a bid, and so on. The algorithms are dependent on the agent platform but should be independent of the application domain and the base platform. We structure this layer as a set of basic entities that model trading agent interactions as they exchange goods in a market. These abstract entities are the building blocks of the Catalactic algorithms.

*Peer agent layer.* Platforms that host the Catalactic agents offer a generic P2P capability for the discovery and communication mechanism, and a programmable interface with the underlying platform. This layer covers the basic functions that all implementations will use; it's responsible for interfacing with the underlying platform and extending it where necessary.

*Base platform layer.* This layer, which might be domain specific, supports applications and Catalactic middleware. The model of interaction with the Catalactic middleware depends on the base platform's architecture but generally requires that developers implement a connector to route resource requests to the corresponding economic agents. In some cases, developers might have to extend core platform components, such as the GRAM (Globus Resource Allocation Manager).<sup>6</sup>

### **Middleware toolkit selection**

To select our middleware toolkits, we considered three different but related factors: potential application scenarios, software architecture, and available middleware toolkits. On the basis of these requirements, we selected and reviewed six toolkits: the DIET (Decentralized Information Ecosystem Technologies) and JADE (Java Agent Development Framework) agent platforms, J2SE (Java 2 Platform, Standard Edition), WSRF/OGSA (WS-Resource Framework/Open Grid Services Architecture), Web services, and JXTA (Juxtapose). We used the CATNETS software architecture to evaluate the toolkits' functional properties<sup>7</sup> and evaluated their technical characteristics and suitability. Ultimately, we concluded that our criteria required a composition of different middleware toolkits.<sup>7</sup> Specifically, we could achieve performance enhancements using a lightweight agent implementation (as in DIET), interoperability using Web-services-based communication, and scalability by strongly decentralizing key functions (as in JXTA). We implemented CATNETS middleware by combining DIET with JXTA and WSRF/OGSA, thereby achieving a good balance between the functional and nonfunctional requirements.

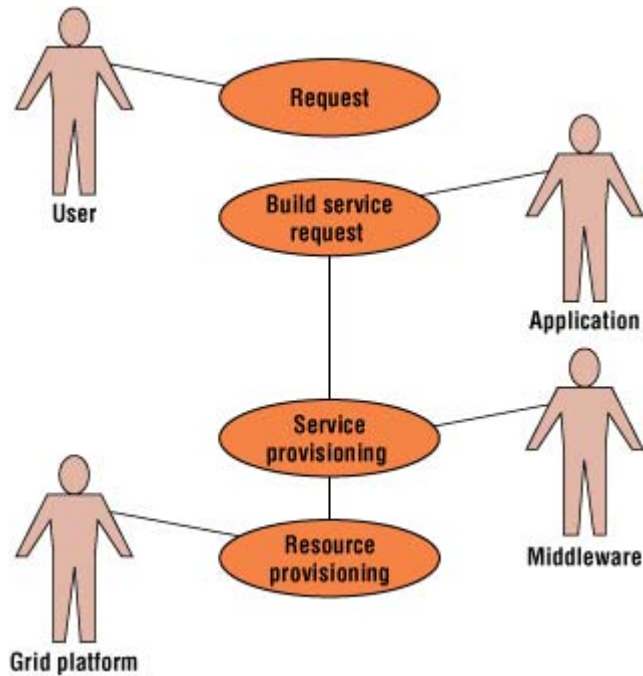
### **Application integration**

In addition to particular application requirements, Catalaxy requires specific placement of logical components to integrate with Catalactic middleware.

### **Catalaxy application requirements**

In the context of a particular problem, different types of applications and users might come together to form a virtual organization. Generally, people establish a VO to combine resource providers' expertise to solve a single problem. We can view a VO as an abstraction for grouping resource providers in a particular project context for a particular duration. Grid infrastructure lets VO users interact until they solve the problem or the VO administrator decides to terminate it. Organizers can use market-based coordination to identify possible VO participants on the basis of policy adherence and service-level objectives. Such VOs might include resources such as computing systems connected over the Internet. Each VO participant could also try to maximize its own utility within the market. There, coordination mechanisms have manifold requirements because the applications and users have different requirements for both auction mechanisms and associated underlying institutional rules.

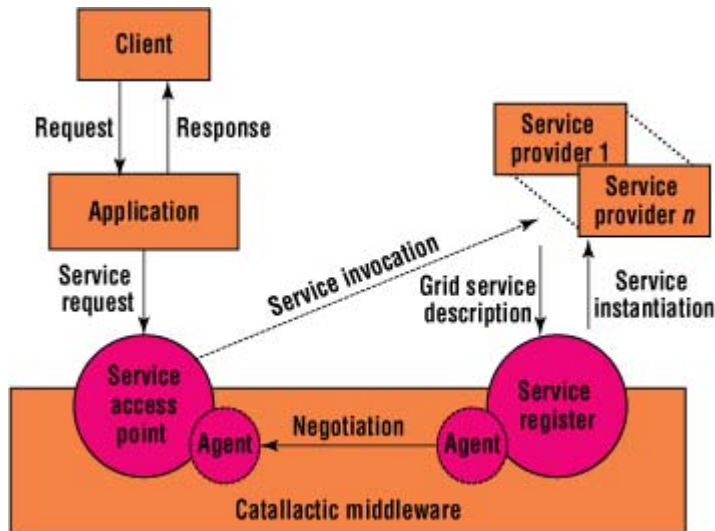
Figure 2 shows a view of actors in a generic application prototype. The user accesses an application and makes a task-execution request. The application handles the request, which transforms the user request into an application-specific task. To fulfill this request, the middleware must provision one or more services; this, in turn, requires the provisioning of a resource bundle managed by the Grid platform.



**Figure 2. Application use case. The user accesses an application and requests a task execution, which the application transforms into an application-specific task.**

### Application interaction

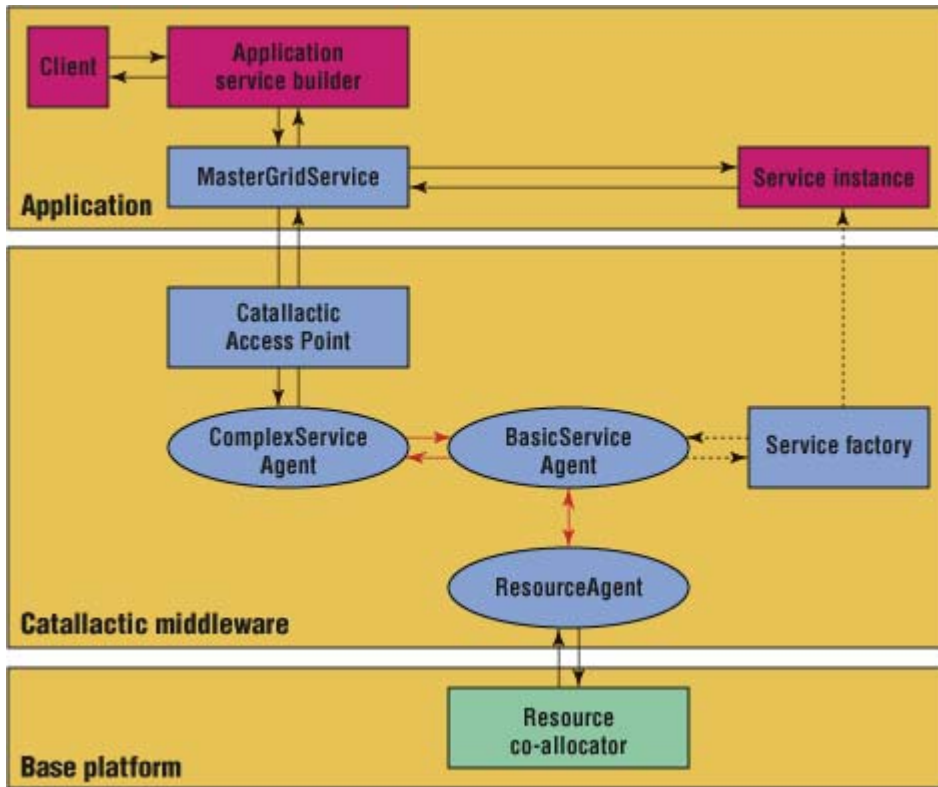
Figure 3 illustrates the interaction between an application and the middleware. When a client issues a request, the application determines which services are required to fulfill it. The middleware searches among available service providers who have registered their service specifications, including contractual conditions, policies, and quality of service. When the application (the buyer) finds a suitable service provider, it negotiates requirements with agents who act as sellers on service providers' behalf. Once the trading agents reach an agreement, the service factory creates a service instance for the application to use.



**Figure 3. Application integration. The application determines which registered services meet the services specifications that fulfill the client's request. It then negotiates with a service provider's agent to reach an agreement.**

## Integrating Grid applications with the Catalactic middleware

Figure 4 shows a detailed architectural view of this integration and identifies logical component placement along the application, Catalactic middleware, and base platform layers. To interact, the application needs an interface to issue requests for services to the middleware and reference the service instances it returns. For a general application to use the underlying Grid market middleware, it requires the Master Grid Service (MGS) module and the Catalactic Access Point (CAP) module.



**Figure 4. Architecture for integrating Catalactic middleware and Grid applications. Logical components are placed along the application, middleware, and base platform layers.**

*The MGS module.* This module provides an interface between the application and the Catalactic market's access points. The MGS is like a service interface; any application that wants to trade on the market must implement it. Conceptually, this module is generic. However, to implement it, developers must configure it with application-specific logic, such as the budget for purchasing services and the services' particular requirements. The MGS is connected to the Catalactic access points, which are advertised within market registries. To connect to the market, application users must find the nearest registry (in terms of their IP domain or connection latency).

*The CAP module.* Applications need this module to interact with a Catalactic market. We have implemented the CAP as a Web service that interacts with the application via the MGS. The CAP provides an entry point for connecting to the Catalactic middleware to identify the list of complex services available on markets. It also offers a way for the middleware and application to exchange service-level agreements (templates and offers) as follows:

- `getAgreementTemplate()` provides access to the agreement template hosted by the CAP. Developers can implement the repository via a database or file system. The template identifies parameters that the CAP can understand and requires the requesting application to use those parameters. We thereby avoid the need for a CAP to understand any application-specific parameters.

- `receivedAgreementOffer()` receives the agreement offer, which contains the application's requested parameters. These parameters form the basis for discovering a suitable service on the market.

At the middleware layer, a set of agents negotiates for services and the resources necessary to execute them:

- A *complex service agent* acts on the application's behalf to initiate the negotiation.
- *Basic service and resource agents* manage the negotiation for services and resources, respectively.

The middleware layer also provides a *service factory* to create a service instance on the target execution platform during negotiation. Finally, at the base platform layer, a local resource manager manages service-instance allocation to resources. The resource represents the service's "state" from the middleware's perspective (the service isn't necessarily stateful from the application's perspective).

Figure 5 shows in detail the events sequence in the interaction between applications and the Catalytic Grid middleware. The application requires service and resource allocations that aren't necessarily co-located; therefore, it must make a request to an access point/middleware in which negotiations take place to find and access them.

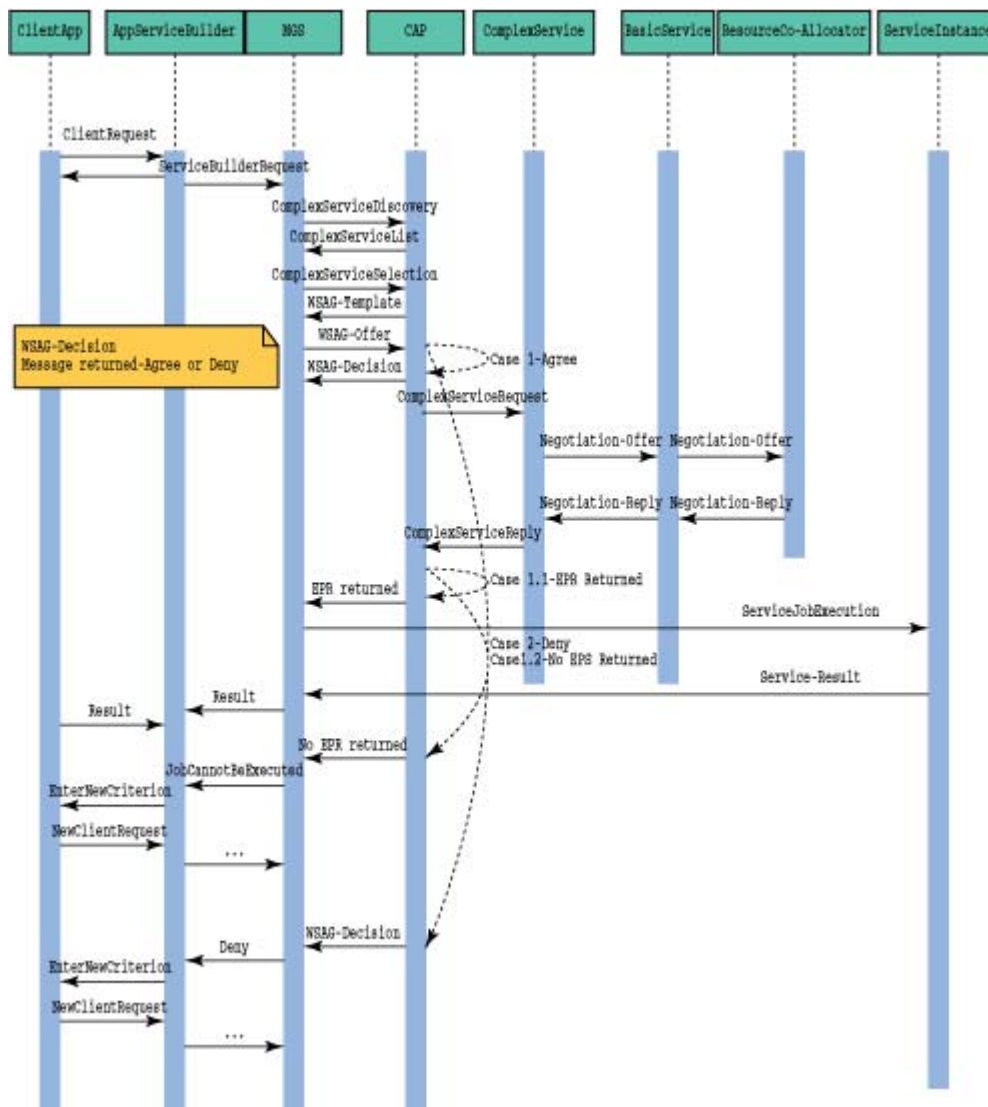


Figure 5. Interaction sequence between applications and the Catalytic Grid middleware.

We can summarize the information flow among the logical components as follows. First, a client issues a request to the application via the application service builder (`AppServiceBuilder`) module—the invoked application instance that interprets the client request into a service. This module then sends the MGS a request for service execution. (These two modules are part of the application box in figure 3). Next, the MGS contacts a service access point (a CAP) asking for a WS-Agreement template for such a service. The MGS fills in the template and sends back an agreement offer.

The complex service agent initiates Catalactic mechanisms to find appropriate basic services and resources. To locate basic service agents, the complex service agent uses discovery mechanisms implemented in the middleware's peer agent layer. After discovering several basic service agents, the complex service agent selects one and begins negotiations. In turn, each basic service agent must discover and negotiate with resource agents in the resource market to find resources on which to execute the service. The economic framework layer (see figure 1) implements negotiations using different protocols, depending on the agent's strategy.

When the complex service agent reaches agreement with a basic service agent, the resource agent instantiates a resource to track the allocated resources, then returns a handle for the resource to the basic service agent. Basic service agents then use the service factory to create an instance of the service on the selected GT4 (Globus Toolkit 4) container. The basic service agent subsequently sends the complex service agent a reference of this service instance and the related resources it uses. Finally, the CAP returns the service reference to the MSG, which uses it to invoke the service to be executed.

## Prototype implementation

We now offer details on how we implemented our prototype.

### Cat-COVITE application

To illustrate the use of the Catalactic middleware, we extend an existing application developed in the Collaborative Virtual Teams (COVITE) project.<sup>4</sup> The application is primarily intended for the architecture-engineering-construction industry, where suppliers of building products (such as doors, windows, and light fittings) and purchasers (building developers and construction companies) collaborate to procure a particular construction project's supplies. Because suppliers make their product catalogs accessible as Web services, a single purchaser can query multiple catalogs concurrently.

In this application, participants from several organizations must collaborate because purchasers can also act as suppliers to larger construction companies and traditional suppliers must work with industry associations who develop the schemas that suppliers use to advertise their products. These interactions happen concurrently, requiring real-time collaboration among geographically remote individuals. Each consortium is in effect a VO. The application permits searching across multiple supplier databases to retrieve products that match purchasers' and contractors' criteria. The application facilitates this search by using multiple networked machines to retrieve the matching products.

We divide the COVITE prototype application into two functional services: security service and multiple-database search service. The MDSS enables searching across numerous supplier databases using a single MGS instance, which might in turn interact with a search service hosted on a networked machine cluster (with one search service per machine). We define the search query according to a given construction project's specific data model. We don't allow arbitrary text queries (as in the Google search engine, for example).

The COVITE application uses a centralized MDSS and lets VOs plan, schedule, coordinate, and share services between designs and from different suppliers. We extended the application prototype to use a Catalactic market model as a way to discover suitable services. Our Catalactic Collaborative Virtual Teams (Cat-COVITE) prototype is based on a service-oriented architecture and consists of three main elements:

- one or more user services,

- an MGS that interacts with the Catalactic middleware to find a service instance's end-point reference, and
- one or more service instances that are being hosted on a particular resource.

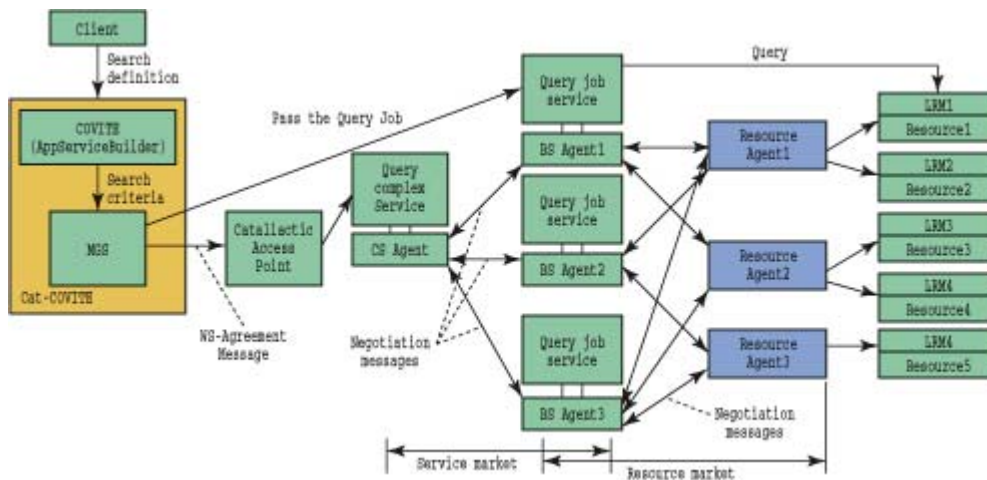
The Cat-COVITE prototype searches through distributed product catalogs—modeled as a Web-services-enabled database—using a distributed search via multiple machines within a cluster.

### Catalactic model for Cat-COVITE scenario

Figure 6 shows a Cat-COVITE prototype instance and related Catalactic agents as buyers and sellers in the service and resource markets. Cat-COVITE uses COVITE application components, which we can map to Catalactic market actors. The COVITE prototype's main entity is the MGS, which uses services from a central broker to execute an application task. In contrast, Cat-COVITE uses the CAP entity to find those services on decentralized markets. The Cat-COVITE instance's entities are

- the complex service, which is an abstract component consisting of the MGS, the CAP between the application and the market, and the complex service agent;
- the query job service, which is a type of basic service; and
- the resource where the query service executes, which is a type of computational resource.

We don't consider service composition mechanisms; we concentrate on each basic service's allocation process.



**Figure 6. Catalactic COVITE prototype. Cat-COVITE uses the Catalactic Access Point (CAP) entity to find services on decentralized markets.**

The complex service agent is the service market's buyer entity. The query job service—via the basic service agent—is the service market's seller entity. To begin, the complex service agent starts parallel negotiations with several agents representing query job services (basic services) and chooses one on the basis of price negotiation. Concurrently, the complex service, via MGS, translates a request to a basic service (such as query job service).

Query job service involves query execution on a particular database and consists of a

- query job execution environment for deploying the query service over cluster machines that can execute the query, and
- translation of a query to resource requirements.

Resource-seller entities can provide a set of resources via the local resource manager accessed via resource agents. The query job service is the resource market's buyer entity and the LRMs are its seller entities. A basic service agent's main function within the resource market is co-allocation of resource bundles through parallel negotiation with different resource providers (LRM entities).

### **Physical deployment on GT4 containers**

We can implement the previous section's logical architecture in different ways depending on the base platform. We've implemented it on a GT4-based platform with the following assumptions:

- The application-specific services were predeployed on a set of GT4 containers.
- The only "resources" considered in the negotiation are the "rights" to execute the service on a specific container.
- The service can be instantiated on a container using a generic factory.

The application and the MGS middleware interface are co-located with the complex service agent, which represents the application in the negotiation process. The corresponding basic service agent resides on each GT4 container deploying the service and negotiates with the complex service agent for service access. The GT4 containers also host a resource agent that negotiates with the basic service agent for the rights to execute the service in this container. Finally, as a result of the negotiation process, the resource is allocated, representing the rights to execute the service in this container.

### **Middleware implementation**

To build this implementation, we used four middleware toolkits:

- the DIET agent platform,<sup>5</sup> which provides a modular, lightweight, and scalable execution platform for agents;
- JXTA, which offers a peer-to-peer platform; and
- the WSRF/OGSA implementation of GT4, which offers full support for resource management in distributed, service-based environments.

We implemented the middleware as a set of simple, specialized agents. Framework agents support the basic functions for implementing economic algorithms, including markets access, negotiation, and good trading. To support system execution, peer-agent-layer agents implement low-level functionalities: overlay network, object discovery, and communication. We use JXTA protocols to route messages among agent nodes and thus create an overlay network for object discovery and communication.<sup>8</sup> We base our local resource management—in this case, services offered by the service providers—on GT4's WSRF. We describe our middleware implementation in detail elsewhere.<sup>9</sup>

We use the JXTA Peer Resolver Protocol to implement the decentralized mechanism that propagates search requests, allowing complex, multi-attribute queries using each node's registered query resolvers. The query resolvers use GT4's index service to resolve queries against the specific search attributes and locate the desired services and resources.

### **WS-Agreement in Cat-COVITE prototype**

The WS-Agreement protocol specification, developed by the Global Grid Forum's GRAAP (Grid Resource Allocation and Agreement Protocol) Working Group, is an XML protocol for specifying an agreement between a resource/service provider and a consumer.<sup>10</sup> WS-Agreement generally aims at a one-shot interaction and isn't directly intended to support negotiation. However, it can form a useful basis for conducting negotiation between two parties and choosing between multiple service and resource providers. In this case, the service provider acts as the agreement provider, while the service consumer acts as the agreement initiator. A WS-Agreement consists of

- the (optional) agreement name;

- the agreement context, which includes the parties to an agreement, reference to the provided services, and the agreement's lifetime;
- the agreement terms, which describe the agreement itself and might contain service description terms, which provide information needed to instantiate or otherwise identify a service to which this agreement pertains; and
- the guarantee terms, which specify the service levels that the parties are agreeing to.

In the Cat-COVITE application, the WS-Agreement use scenario is as follows. First, an MGS has to run a query job as an application job and send the CAP an agreement offer to find a query service based on the agreement template (downloaded from the CAP). The complex service agent—acting on behalf of the CAP-selected complex service (the MGS)—negotiates with the basic service agents in the CATNETS environment for query services to fulfill the job. The agreement template specifies the service description elements that the advertising factory allows. The “Cat-COVITE Agreement Templates” sidebar shows a WS-Agreement-compliant agreement template for Cat-COVITE applications. The agreement initiator (in this case, the MGS) initiates the agreement offer (see the sidebar for an example; it, too, is WS-Agreement compliant). The agreement provider either accepts the offer's conditions, by initiating an agreement acceptance (which is the same as the agreement offer), or rejects the offer, in which case the agreement initiator must send an alternative offer. This continues until a consensus is reached or until the initiator or provider terminates the interaction. In this scenario, the negotiation between the agreement provider and initiator is price-based.

SIDEBAR: Cat-COVITE Agreement Templates

[ ]

Following is an example of an agreement template that agreement providers use for the specific query complex service available on the Catalactic market:

```
<?xml version="1.0" encoding="UTF-8"?>
<wsag:AgreementTemplate AgreementID="QueryTemplate-v001"
xmlns:wsag="http://schemas.ggf.org/graap/2005/09/ws-agreement">
  <wsag:Name>QueryComplexService</wsag:Name>
  <wsag:Context>
    <wsag:AgreementInitiator>
      <!--can be a URI or a security identity of the initiator -->
      NameOfTheInitiator
    </wsag:AgreementInitiator>
    <wsag:ExpirationTime>DateTime</wsag:ExpirationTime>
    <wsag:TemplateID>QueryTemplate-001
    </wsag:TemplateID>
    <wsag:TemplateName>QueryComplexService
    </wsag:TemplateName>
  </wsag:Context>
  <wsag:Terms>
    <BasicServiceName>QueryBasicService
    </BasicServiceName>
    <NumberOfBasicServiceNodes>
    </NumberOfBasicServiceNodes>
    <BasicServiceConstraints>
      <BasicServiceType>
      </BasicServiceType>
    </BasicServiceConstraints>
    <Price>
    </Price>
```

```
</wsag:Terms>
</wsag:AgreementTemplate>
```

Following is an example of an agreement offer that agreement initiators use for the specific query complex service request:

```
<?xml version="1.0" encoding="UTF-8"?>
<wsag:AgreementOffer AgreementID="QueryOffer-v001"
xmlns:wsag="http://schemas.ggf.org/graap/2005/09/ws-agreement">
  <wsag:Name>QueryComplexService</wsag:Name>
  <wsag:Context>
    <wsag:AgreementInitiator>
      <!--can be a URI or a security identity of the initiator -->
      NameOfTheInitiator
    </wsag:AgreementInitiator>
    <wsag:ExpirationTime>DateTime</wsag:ExpirationTime>
    <wsag:TemplateID>QueryTemplate-001</wsag:TemplateID>
    <wsag:TemplateName>QueryComplexService
    </wsag:TemplateName>
  </wsag:Context>
  <wsag:Terms>
    <BasicServiceName>QuerygBasicService
    </BasicServiceName>
    <NumberOfBasicServiceNodes>1
    </NumberOfBasicServiceNodes>
    <BasicServiceConstraints>
      <BasicServiceType>
        Architectural/Engineering/Construction
      </BasicServiceType>
    </BasicServiceConstraints>
    <Price>100
    </Price>
  </wsag:Terms>
</wsag:AgreementOffer>
```

## Negotiation protocol

For this prototype's economic agents, we use the ZIP (Zero Intelligence Plus) agents,<sup>8</sup> which use a gradient (ascent/descent) algorithm to set resource prices. A complex service first initiates negotiations by sending to basic services a bid lower than the available budget. If a basic service accepts this bid, it responds by sending a conformity message and allocates the resources it requires to fulfill the request. If a complex service is

unable to buy at that price, it increases its bid until it either wins or reaches the budget limit.

Basic service agents start selling the service at a price that's influenced solely by the price that the executing node's resource agent reports. As basic service agents negotiate, demand for the service also influences its price. If a service agent is selling its service, it incrementally increases the price to determine the market value. When it's no longer able to sell, it lowers the price until it's competitive again or it reaches a minimum price defined by current resource utilization.

The basic service's price update follows a gradient algorithm. If the current price exceeds all bids' maximum offered price, the basic service tries to lower its price to become competitive. If the price is below all the bids, it evaluates the market by raising the price to obtain higher profits.

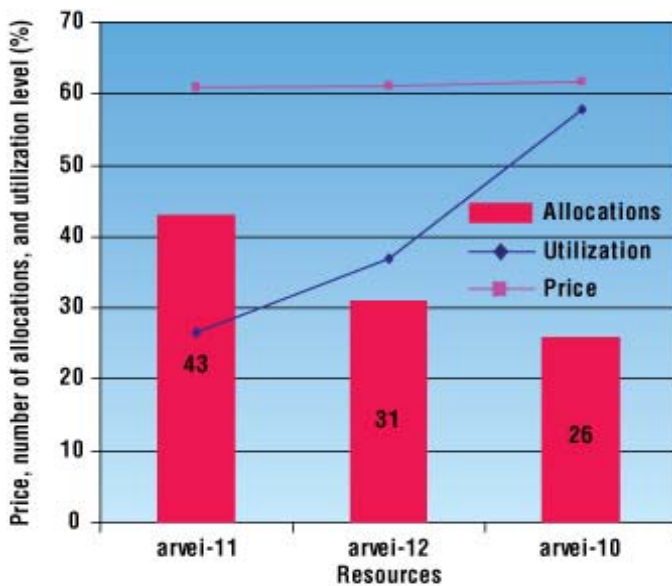
We intentionally limited the prototype's LRM because we wanted to evaluate how the economic mechanism alone impacts resource allocation. The LRM tracks only CPU utilization (using information that the Linux kernel provides) and "allocates" it without any warranty because services compete with other workload for the CPU. Finally, the resource agents calculate resource price on the basis of current resource utilization, as reported by the LRM.<sup>11</sup>

### Prototype performance assessment

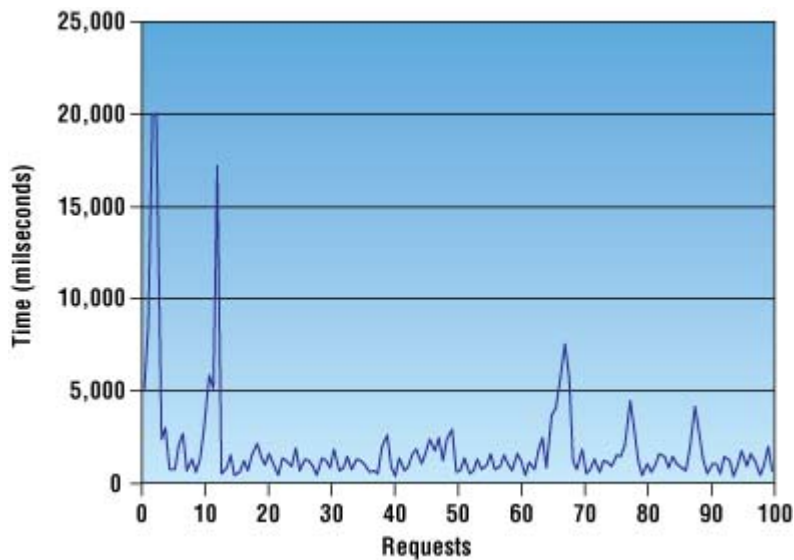
To test the market-based resource allocation mechanism's performance, we set up experiments deploying several middleware instances on a Linux server farm. Each node had dual-CPU Intel PIII, 1-GHz processors with 512 Mbytes of memory. An internal Ethernet network operating at 100 Mbits connected the nodes. We deployed the Catalactic middleware and the application services on three nodes (arvei-10, arvei-11, and arvei-12). We generated a baseline load on all three nodes of approximately 25 percent of CPU usage to simulate background activity.

We launched three application clients concurrently from three other nodes. Each client performed 50 requests with 10-second intervals. Whenever a client won a bid for a service, it invoked the service on the selected node. Each experiment ran for about 10 minutes. To test the algorithm's adaptability to system workload changes, we artificially varied the workload on one of the nodes (arvei-10) up to 95–100 percent of CPU usage.

As Figure 7 shows, the algorithm effectively balanced workload on the basis of price. The reason for this is that the utilization level impacts the price of the server's resources; during negotiation, clients prefer the lowest-priced services. Figure 8 shows how the negotiation time dropped dramatically as agents learned how to set prices to improve their market competitiveness. When market conditions changed, however, the negotiation time increased as agents adjusted to the new conditions.



**Figure 7. Load balancing of resources. The algorithm effectively balanced workload on the basis of price.**



**Figure 8. Negotiation time.** As agents learned how to set prices to improve competitiveness, negotiation time dramatically decreased, although it increased when agents had to adjust to changing market conditions.

Our prototype implementation shows how we can adapt existing applications to use the Catalactic middleware. We thereby provide a resource allocation mechanism capable of both load balancing and adaptation to changing environments.

To handle bargaining process complexities, however, we'll have to extend our negotiation protocol beyond the current WS-Agreement specification. Also, WSRF specifications are still too general and don't offer a clear approach for managing virtual resources. Using standard specifications—such as WSRF and WS-Agreement—offers the main incentives for other application users to employ our Catalactic market implementation. By providing interaction with the market via an access point (the CAP), we ensure that developers won't need to internally modify their applications. In addition, unlike other Grid marketing models that use centralized brokers, the Catalactic market model's peer-to-peer nature makes our approach suitable to distributed decentralized systems, such as computational and data grids.

The approach we adopted in Cat-Covite can apply to other industrial applications that use distributed databases. Consequently, the lessons we learned with this application and its integration with the Catalactic middleware could be useful to a wide community.

## Acknowledgments

The European Union partially supported our work under Contract CATNETS EU IST-FP6-003769. This is an extended version of an article we presented at MGC2005 (Middleware for Grid Computing) workshop.

## References

1. R. Buyya et al., "Economic Models for Resource Management and Scheduling in Grid Computing," *J. Concurrency and Computation: Practice and Experience (CCPE)*, May 2002, pp. 1507–1542.
2. F.A. Hayek et al., *The Collected Works of F.A. Hayek*, Univ. of Chicago Press, 1989.
3. O. Ardaiz et al., "The Catalaxy Approach for Decentralized Economic-Based Allocation in Grid Resource and Service Markets," Special Issue on Agent-Based Grid Computing, *Int'l J. Applied Intelligence*, vol. 25, no. 2, 2006, pp. 131–145.
4. L. Joita et al., "Supporting Collaborative Virtual Organisations in the Construction Industry via the Grid," *Proc. UK e-Science All-Hands Meeting*, 2004; [www.allhands.org.uk/2004/proceedings/papers/182.pdf](http://www.allhands.org.uk/2004/proceedings/papers/182.pdf).
5. T. Eymann et al., "Catalaxy Based Grid Markets," *J. Multiagents and Grid Systems*, vol. 1, no. 4, 2005, pp. 297–307.

6. Foster, "A Distributed Resource Management Architecture that Supports Advance Reservations and Co-Allocation," *Proc. Int'l Workshop on Quality of Service*, Globus, 1999; [www.globus.org/alliance/publications/papers/iwqos.pdf](http://www.globus.org/alliance/publications/papers/iwqos.pdf).
7. CATNETS project deliverable, "D3.1: Implementation of Additional Services for the Economic Enhanced Platforms in Grid/P2P Platform: Preparation of the Concepts and Mechanisms for Implementation," Sept. 2005, [www.iw.uni-karlsruhe.de/catnets/fileadmin/publications/WP3-DeliverableC\\_1.pdf](http://www.iw.uni-karlsruhe.de/catnets/fileadmin/publications/WP3-DeliverableC_1.pdf).
8. C. Preist and M. van Tol, "Adaptive Agents in a Persistent Shout Double Auction," *Proc. 1st Int'l Conf. Information and Computation Economies (ICE 98)* ACM Press, 1998, pp. 11–18.
9. O. Ardaiz et al., "An Architecture for Incorporating Decentralized Economic Models in Application Layer Networks," *Smart Grid Technologies Workshop*, 2005.
10. *Web Services Agreement Specification (WS-Agreement)*, Grid Resource Allocation and Agreement Protocol Working Group, Global Grid Forum, 28 June 2005.
11. C.S. Yeo and R. Buyya, "Pricing for Utility-Driven Resource Management and Allocation in Clusters," *Proc. 12th Int'l Conf. Advanced Computing and Communication (ADCOM 2004)*, Allied Publisher, 2004, pp. 32–41.



**Liviu Joita** is a research associate at the School of Computer Science, Cardiff University. His research interests include Grid computing, Grid economies, Web services, distributed computing, security, collaborative work in virtual organizations, agents, and multiagent systems. He received an MS in managerial and technologic engineering from the University of Polytechnics, Bucharest, and an MS in information and communication technology from International University in Germany. He is the cochair of CCGrid's International Workshops on Agent-Based Grid Computing for 2006 and 2007. Contact him at the School of Computer Science and the Welsh eScience Centre, Cardiff Univ. Queen's Buildings, 5 The Parade, Roath, Cardiff, CF24 3AA, UK; [l.joita@cs.cardiff.ac.uk](mailto:l.joita@cs.cardiff.ac.uk).



**Omer F. Rana** is a reader at the School of Computer Science and the deputy director of the Welsh eScience Center at Cardiff University. His research interests include high-performance distributed computing, multiagent systems, and data mining. He received a PhD in neural computing and parallel architectures from Imperial College of Science, Technology and Medicine (London University). Contact him at the School of Computer Science and the Welsh eScience Centre, Cardiff Univ. Queen's Buildings, 5 The Parade, Roath, Cardiff, CF24 3AA, UK; [o.f.rana@cs.cardiff.ac.uk](mailto:o.f.rana@cs.cardiff.ac.uk).



**Pablo Chacín** is a PhD student at the Technical University of Catalonia, Barcelona. His research interests include autonomic computing, economic-based resource allocation, grid middleware, and large-scale distributed systems. He received a master's degree in information technology management from the

Instituto Tecnológico y de Estudios Superiores de Monterrey, Mexico. Contact him at the Computer Architecture Dept., Technical Univ. of Catalonia, Jordi Girona 1-3, Campus Nord D6, Barcelona 08035, Spain; pchacin@ac.upc.edu.



**Isaac Chao** is a PhD student at the Technical University of Catalonia, Barcelona. His research interests include multiagent systems and agent-based coordination applied in Grid resource allocation and complex networks. He received a BS degree in physics from the University of Santiago de Compostela, Spain. Contact him at the Computer Architecture Dept., Technical Univ. of Catalonia, Jordi Girona 1-3, Campus Nord D6, Barcelona 08035, Spain; ichao@ac.upc.edu.



**Felix Freitag** is a full-time adjunct lecturer in the Computer Architecture Department of the Technical University of Catalonia, Barcelona. His research interests include distributed computing and performance measurement of parallel and distributed systems. He received a PhD from the Technical University of Catalonia in Telecommunication Engineering in 1998. Contact him at the Computer Architecture Dept., Technical Univ. of Catalonia, Jordi Girona 1-3, Campus Nord D6, Barcelona 08035, Spain; felix@ac.upc.edu.



Leandro Navarro is an associate professor in the Computer Architecture Department of the Technical University of Catalonia, Barcelona. His research interests include the design of scalable and cooperative Internet services and applications. He received a PhD in telecommunication engineering from Polytechnic University of Catalonia, Spain. He is a member of the IEEE Computer Society, the IEEE Communications Society, ACM (SIGGroup, SIGComm), APC (Association for Progressive Communications; council member), IFIP TC6 WG6.4, and CPSR (Computer Professionals for Social Responsibility). Contact him at the Computer Architecture Dept., Technical Univ. of Catalonia, Jordi Girona 1-3, Campus Nord D6, Barcelona 08035, Spain; leandro@ac.upc.edu.



Oscar Ardaiz is an assistant professor in the Public University of Navarra, Spain. His research interests include application overlays and collaborative systems based in Grid technology. He received his PhD from the Polytechnic University of Catalonia. He cochaired CCGrid's International Workshop on Collaborative and Learning Applications of Grid Technology in 2004 and 2005. Contact him at the Dept. of Mathematics and Informatics, Public Univ. of Navarra; Campus de Arrosadia, Pamplona 31006, Spain; oscar.ardaiz@unavarra.es.

#### **Related Links**

- [DS Online's Middleware Community](#)
- ["Agents, Grids, and Middleware," IEEE Internet Computing](#)
- ["Self-Organizing Resource Allocation for Autonomic Networks," 14th Int'l Workshop Database and Expert Systems Applications](#)

#### **Cite this article:**

Liviu Joita, Omer F. Rana, Pablo Chacin, Isaac Chao, Felix Freitag, Leandro Navarro, and Oscar Ardaiz, "Application Deployment on Catallactic Grid Middleware," *IEEE Distributed Systems Online*, vol. 7, no. 12, 2006, art. no. 0612-oz001.