地理學論叢 第30號(1997. 8), 1-18.

A Study on Coarse-grained Service Object Model for GIS Interoperation

Key-Ho Park*

Abstract : The OMEGA system is intended to support geoscientific modeling tasks. In this paper, we emphasize those aspects of OMEGA that are relevant to the integration of distributed computational tools and objects. OMEGA is capable of delegating specialized computations to a variety of external packages through the SERVER objects and associated interface methods. In this sense, OMEGA may be viewed as a software coupling system into which existing packages and application-specific softwares may be plugged with minimal effort. Most of the engineering details of the coupling mechanism are encapsulated and automated for the end-user.

Key Words: GIS, object-oriented, distributed, interoperability, client/server, service object model

요약 : 본 연구에서는 지리정보가 동원되는 수치모델링 작업을 지원하는 환경으로서 객체지향 OMEGA 시스템을 제시한다. 특히 네트워크상에 분산되어 있는 자료와 분석도구의 통합기능에 초점을 두어 지리정보시스템의 상호운용성 제고를 목적으로 시스템 구조를 분석하고, 각 컴포넌트별 특성과 구현기법에 관하여 논한다. OMEGA 시스템은 공간정보분석과 연산작업을 위해 고안된 다양한 외부 소프트웨어를 호출하여 처리할 수 있도록 설계하였다. 이를 위해 외부 소프트웨어를 객체지향이라는 틀에서 SERVER라는 객체 클래스와 클래스 메소드를 통해 인터페이스로 재구성하였다. OMEGA 시스템의 구성요소로서 객체화가 시도된 소프트웨어는 그 기능별로 (1)형식논리에 기반을 둔 "rule" 객체와 이를 이용하는 "추론서버", (2)영상처리를 전담하는 서버객체, (3)지리정보시스템 객체, 그리고 (4)새로운 서버객체의 관리를 위한 메타서버 객체 등을 포함한다. 지금까지 객체화에 관련된 연구가 그 대상을 '데이터'에 국한시킨데 비하여 본 연구에서 제안된 OMEGA 시스템은 데이터의 처리라는 '프로세스'에 대한 객체화 작업을 수행할 수 있고 이를 지원하기 위한 이론적 기술적 토대를 갖추고 있다. OMEGA 시스템은 최근 지리정보시스템의 연구동향인 "플러그-인" 구조와 "Componentware"를 위한 하나의 접근방식을 제시하고 있으며 소프트웨어 연계(Coupling)를 위한 기반 미들웨어를 포함하고 있다. 더욱 중요한 것은 지리정보시스템 자체가 하나의 서비스 객체로 모형화됨으로써 이질적인(heterogeneous) 지리정보시스템간의 상호운용성이 제고된다.

주요어: 지리정보시스템, 객체지향, 분산형, 상호운용성, 클라이언트/서버, 서비스 객체모델

I. INTRODUCTION

We present a new object-oriented system, OMEGA (*O*bject-based *M*odelling *E*nvironment for *G*eoscientific *A*pplications), which is focussed on supporting data-intensive modeling activities in geoscience. Such activities typically make use of a large array of software tools. In this paper, we emphasize those aspects of OMEGA that are relevant for the integration of distributed, autonomous, and interoperating computational tools and objects.

* Assitant Professor, Department of Geography, College of Social Sciences, Seoul National University, Seoul Korea.

A computational environment which supports geoscientific modeling must facilitate the highly interactive and iterative processes of constructing and manipulating algorithms and models, as well as large and complex datasets(Smith et al., 1995). In designing and implementing the OMEGA system(Park, 1994), we have addressed the issues involving the development and integration of systems capable of the various levels of abstraction of data and associated meta-procedures envisioned in the EOS Data and Information System (Asrar and Dokken, 1993). A prototype of OMEGA has been implemented and fully operational.

The computational environment for geoscientific studies is a comprehensive system which includes, in the least, the following components:

① a rich data model accommodating a wide spectrum of complex objects, ② language support for data definition and manipulation, ③ a powerful and efficient programming language, ④ a persistent storage management system and query language, ⑤ an application development environment integrated with specialized computing tools, ⑥ friendly user interfaces.

Existing information systems do not provide adequate support for higher level abstraction and manipulation of computational modules for geoscientific modeling activities. To facilitate dynamic configurations of a user's computational environment, we propose a computational server management system in which not only data but also computations are abstracted, organized, and accessed at the level of 'server'. By the generic term 'server', we will refer to the tools and packages that geoscientists employ for their research. An abundance of special purpose systems and tools is available today: image processing and data visualization, symbolic computation for logic and mathematical equations, statistical packages, to name a few. Most of the current systems employed in geoscientific research are either ad hoc integrations of components, or monolithic systems that lump a large amount of functionalities into a single module. That is, the system architecture has either too little or too much structure, which results in either absence of or duplication of capabilities. Development of a computation environment should not involve re-inventing wheels. Instead, a system should be dynamically extended and configured based on together time-proven software packages that scientists use. Systems of open and modular architecture are easier to modify and extend. The coupling of various components of the system must be loose enough that they may be replaced with new, experimental, or custom-made pieces. In this regards, we observe the lack of well-founded abstractions for system components and high-level protocols for the intra-system interfaces.

Ⅱ. Overview of OMEGA

The goal of OMEGA is to build a better computing environment for geoscientific researchers (referred to as OMEGA 'users'). The scope of OMEGA includes a conceptually rich and uniform data model, an expressive user interface language, specifications and methodologies for manipulating spatial objects, a variety of value-added system abstractions, and implementations techniques that bridge theory and practice. OMEGA embodies a new data model integrated with novel abstractions as well as features drawn from existing data models. In OMEGA, we have refined the notion of 'object' by abstracting an object as a 'server,' and encapsulating interface methods of the object as its 'services.' A particular server(object) has a name(Oid) to which users submit their requests. Requests from clients are valid if they are among the set of services publically advertised by the server. A client's request(method selection) is handled within the server by dispatching (method invocation) the appropriate service routine (method implementation). When the execution of the routine has completed, the server delivers the results to the client. The main rationale for an encapsulated service-based interface of objects is that the conceptualization and processing of data queries and transactions become clean and simple because of the uniform protocols of client-server interactions.

The clients interact with a server in a fundamentally services are desired, but not the services are to be processed inside the server.

OMEGA has a user-interface language, SOUL (Structured OMEGA User Language). A user's conceptualizations of an application problem may be represented by the constructs and built-in primitives in SOUL as a basis. SOUL is a multi-faceted language, and an ultimate medium through which users interact with, and harness the underlying computational infrastructure of OMEGA. SOUL, as a computation command language, has an extremely simple syntactic structure, much like the "**select from where**" of SQL. The structure of SOUL expression is a triple "[O,M,P]", where O,M,P represents syntactic blocks that are respectively interpreted (evaluated) to be an object, method, and parameter.

P is optional and may be absent if the method denoted by M requires no trailing parameters. Each of these syntactic units may have nested internal structures, which will eventually be evaluated to be either object, method, or parameters depending on the position within the triple.

SOUL bears many similarities to the method invocation of object oriented programming languages. The differences are in the uniform treatment of attributes and methods. SOUL obviates the dot-notations ('.'), which is used to access object attributes in most object-oriented languages. Unlike most other object-oriented models, OMEGA allows uniform and clean user interfaces to objects by treating attributes as methods. The interfaces of reading and assigning attributes of the OMEGA objects are achieved uniformly in terms of method invocations. While a method may be for efficiency (i.e., a method is a constant function), the issue of whether methods are stored or evaluated is really an implementation issue that is orthogonal to the conceptual uniformity of the method-based interface of the OMEGA objects.

Unlike other extensible database systems that are coupled with external programming languages (e.g., C++), programmability is a native feature of SOUL. Therefore, SOUL is devoid of the overhead incurring from interfacing with external programming languages. Most realistic applications involving arbitrary sequences of computations can be written, modified, extended, and maintained interactively in SOUL.

OMEGA supports persistency of its objects, and is thereby a fully operational database system. OMEGA has built-in classes of spatial objects and their manipulation primitives. OMEGA is capable of delegating specialized computations to a variety of external packages. In this sense, OMEGA may be viewed as a software coupling system into which many existing packages and application-specific software can be plugged with minimal effort. Most of the engineering details of the coupling mechanism are encapsulated and automated for the end-user.

Ⅲ. Abstraction of Computational Packages in OMEGA

OMEGA features a novel conceptualization for integrating various computational packages into a coherent and interoperable computing environment. In particular, the software infrastructure provided in terms of the class permits the encapsulation of arbitrary external, autonomous, and heterogeneous software packages, and their coupling to the extensible OMEGA environment. The **SERVER** objects embody external software components. OMEGA makes **SERVER** available to the user as a built-in class where system-level details in implementing server are encapsulated in the class methods. The underlying idea of the class stems from the conceptualization of autonomous software packages as **SERVER**. The net result of this abstraction is that servers are fully supported as first-class objects in OMEGA. All the internal details of handling server objects (e.g., interprocess communication protocols) are automated behind the scene, and thus transparent to users. The OMEGA user-language, SOUL, is augmented by a small number of primitive methods built in the SERVER class for creating and interacting with server objects. Also, users are provided with several command directives. The interfaces with server objects using the directives are operationally equivalent to the method-based interfaces.

In the following sections, we discuss the general motivation and conceptual framework of **SERVER**, introduce user-level language primitives associated with **SERVER**, and present operational semantics of each of the primitives.

1. Background and Motivation

The primary motivation of explicit incorporation of server objects stems from the observation that geoscientists typically work with a diverse array of computational tools in research. A typical geoscientific computing environment may be configured to include numerical analysis packages for partial and ordinary differential equations, GIS and Remote Sensing tools for processing digital map layers and imagery, statistical packages, and data visualization tools. The current practice of system integration in geoscientific research, however, is based on either partially interacting components or a monolithic system that lumps much functionality into a single module. Hence current practices may result in the lack of, or the duplication of computing capabilities. To remedy the situation, we develop well-founded abstractions for external software packages. We also define logical protocols for the intra-system component interfaces and user-level primitives that encapsulate these protocols.

The main technical challenge involved in the **SERVER** class is to design and implement a software infrastructure that minimizes the 'hacks' of adding new components to an existing environment while preserving the sharing and interoperability of services. Such system integration and interfaces are based on an open and modular architecture, and various components of the system are coupled loosely to facilitate their replacement.

There are system abstractions that make distributed computing and networking more amenable to system programmers, which range from the 'pipe' abstraction of the logical 'file' level, to 'socket', to 'RPC-stub' abstractions of location-transparent 'procedure' level. The **SERVER** primitive described in this paper acheves a higher level of abstraction of logical 'program.' The major benefit of the **SERVER** abstraction and associated primitives is that the tasks of programming system integration and configuration are pushed down to the (even technically naive) end-users without relying on the system administrators. Users can extend and configure their computing environment by creating, creating, installing, andinteracting with diverse software tools with minimum efforts using a few **SOUL** primitives.

2. Conceptual Framework of SERVER

The conceptualization of the 'server' objects that are intended to embody external software packages is deceptively simple, and yet powerful. In the OMEGA data model, every object is abstracted as a logical server. Reversely, we may also conceptualize physical servers(i.e., autonomous packages) as logical objects. The computational capabilities that each package delivers are encapsulated as the methods of the server object.

Note that the notion of OMEGA objects does not have to be modified, and the integrity and uniformity of the OMEGA data model is not undermined in accommodating arbitrary pieces of external softwares as **SERVER** objects. Moreover, the syntactic unformity of SOUL is preserved. Consider the following example of an interactive OMEGA session prompted by '**OMEGA>**':

```
OMEGA> p1 distance p2
OMEGA> grass>r. overlay soil landcover
```

The object 'p1' in the first example is, in essence, a server, and one of the services it can handle is a '**distance**' computation. As such, '**distance**' appended by the trailing parameter 'p2' constitutes a complete request of service submitted to 'p1'.

In the second example, '**grass>**' is **SERVER** object in which a GIS package, GRASS, and its services are abstracted. The particular object name(OID) was chosen as '**grass>**' since it resembles the system prompt when GRASS is used independently. One of the module(services) the GRASS package provides is the raster map overlay command '**r.overlay**'. It requires two parameters of raster maps, which, in this example, are '**soil**' and '**landcover**'. During an independent session of GRASS, a user would have entered exactly the same expression at the GRASS prompt '**grass>**'. The only difference when GRASS is made as a server object and used in OMEGA environment is that users explicitly type in the object name ('**grass>**') just like any other SOUL expressions. Note the obvious parallelism between 'object-method-parameter' and 'server-service-parameter' triples.

3. Architecture of SERVER Object

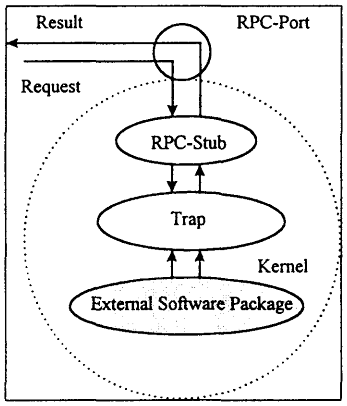
The **SERVER** class and associated primitives are designed and implemented in OMEGA to meet several technical challenges: It should allow easy and seamless accommodation of existing packages. It also should facilitate user's computational environment facilitate incremental extensions (scalability), dynamic and network-transparent configuration of user's computational environment.

Some of the complications involved in importing external software packages into any integrated computing environment are that they are external, autonomous, and heterogeneous. They are written in various external programming languages such as 'FORTRAN', 'C', 'C++' etc. These packages are built without any consideration that they will be used in the OMEGA environment. External packages are autonomous, running independently of other packages. User-interfaces are also very different from package to package, and the run time environments required for these packages are heterogeneous. We believe that the integration of external software should be based on a loosely coupled 'client-server' paradigm, in which the autonomous and heterogeneous nature of the packages are maintained.

The common practice of tight coupling of packages in terms of local interprocess communication is thus insufficient. In this practice, it is the individual subprocesses created by a parent process (through the 'fork' system call) which would run as external programs. The interaction among them relies on local communication 'pipes', and thus will not be suited for a heterogeneous environment spanning over a network. Moreover, the autonomy of each subprocess is not possible because they will terminate as the parent process terminates. This 'master-slave' paradigm of software interaction is quite inferior to the 'client-server' counterpart.

One of the popular approaches to network-distributed objects and interoperable functionalities is the (RPC) facility. We implemented the mechanistics and functionalities of the SERVER class primitives based on RPC/XDR technology. RPC is a mechanism for providing synchronized and type-safe communication between two processes. This mechanism is used for transferring control and data among processes distributed over a number of computers interconnected by a high-speed communication network. The RPC facility, together with the 'external data representation' standards (XDR), allows application programs to make use of distributed services by calling remote procedures by name, and hides the details of underlying layers such as transport protocols, etc.

The basic approach that OMEGA adopts inn integrating a suite of external packages and tools into a single environment is based on the 'virtual



〈Figure 1〉 'Wrapping' External Software Server wrapping' mechanism. As shown in 〈Figure 1〉, the server-wrapping consists of the three logical layers of **Port**, **Stub**, and **Trap**: An interprocess communication poty is allocated for exchanging RPC packets. A stub layer is generated which packs/unpacks data suitable for network transport. A trap layer is provided, which is responsible for translating requests in OMEGA languages into the native commands that the underlying external server understands.

The communication port and stub will be transparent to users and will be automatically allocated at the time of server wrapping. The system also provides a default trap layer which simply passes requests to the underlying package without any translations. Users may write application-specific trap layers.

A trap layer consists of a set of procedures corresponding to each services that a particular server can deliver via the external package. These service routines are viewed as methods associated with the given server object. The external package is considered a kernel in the sense of a UNIX kernel. A trap layer, then, may be viewed as an interface shell which is responsible for interpreting user commands, dispatching service routines stored in the kernel, and delivering results back to the clients.

4. Characteristics of SERVER Object

We now assess the value of our approach to abstracting SERVER objects with respect to the following criteria: (1) the manner and degree to which they extend systems and components, and (2) the degree to which they facilitate distributed computing.

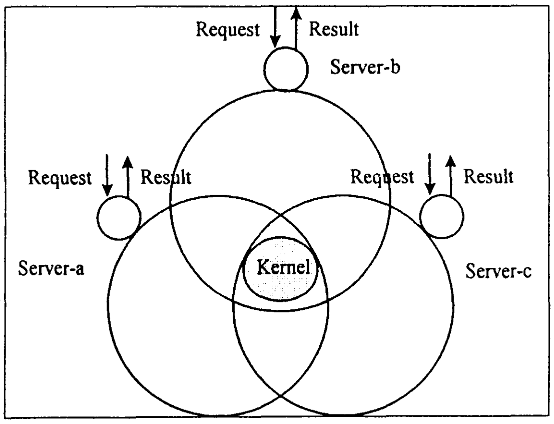
1) Extended computing environment through the SERVER objects

A major benefit incurred from the layered wrapping approach to server construction is that the work required from users to extend their computing environment is greatly simplified. The actual extensions can be observed at two different levels: (1) the system level, and (2) the individual component level. At the level of the system as a whole, the capability of computing environments will be dynamically extended by the integration of external software packages. At the individual component level, the interface of external software will be dynamically extended by modifying trap layers associated with them.

A particular computing environment is a sum total of various component software packages. A notable feature of our approach to system integration is its seamless and dynamic nature. By seamless integration, we mean that external packages are imported as server kernels without any modifications of their internal codes, and thereby its autonomy is preserved. By dynamic integration, we mean that SERVER objects can be created dynamically even while the system is running without any re-compilation or re-linking of modules. External packages are indeed treated very much like dynamically loadable run-time libraries.

We also can observe seamless and dynamic extensibility at the level of each individual SERVER object. The wrapping mechanism is 'virtual' in the sense that each wrapping does not require physical duplication of kernels. The kernels, of course, may be accessed and used as stand-alone as usual. Virtual wrapping of software packages, in effect, simply provides different access paths to them. A single copy of software may be wrapped into any number of server objects at run time. Each different wrapping will result in different interfaces to a single underlying software, because an interface is encapsulated in a trap layer.

〈Figure 2〉 illustrates a case where a single kernel is wrapped into three different versions of **SERVER** objects, each of which provides separate accesses and interfaces. Multiple Versions of Server As such, users merely have to modify a few trap routines in modifying external interface of software; the internal codes of that software remain intact as kernel.



〈Figure 2〉 Multiple Version of Server

2) Distributed computing through the **SERVER** objects

Transparent support for distributed computing is another significant benefit of our approach to creating **SERVER** objects through wrapping. The essential requirements for supporting distributions are location transparency and accommodation of heterogeneity. The physical location in which a particular piece of software resides becomes a non-issue, because the **'port'** layer of a server object encapsulates locational transparency. Interactions with server objects occur across communication networks through the remote procedure calling facility.

The heterogeneity of hardware platforms also becomes a non-issue. Any idiosyncrasies associated with platforms, for which a particular piece of software is compiled, will be localized since requested procedures will be executed in the native platforms regardless of where the requests originate.

Ⅳ. OMEGA Server Pool Management

The idea of the server pool and its management is motivated by the benefits of organizing and accessing servers hierarchically according to their functionalities, locations, and requirements of special hardwares and/or ancillary softwares. By the server pool, we mean the set of servers that are running (or at least able to start on demand) and configured to be accessible by clients. An early logistic design decision we made was to develop a system abstraction of 'meta-servers', which manage the virtual boundaries and sub-boundaries of the server pool.

1. Server Pool and Meta-Server

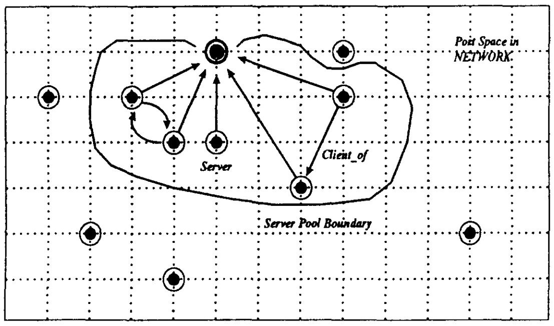
Meta-server plays the role of a central registrar for a given pool of servers active in the OMEGA environment. That is, even after a server is wrapped and put into a network, there should be some mechanism to advertise its availability to potential client processes. Meta-server is envisioned as a centralized directory service of "Who's Who"; in the net-land.

Meta-server also is responsible for logical organizations and managements of a set of servers within the context of the server pool. We define the server pool as the set of servers accessible by clients in the OMEGA environment.

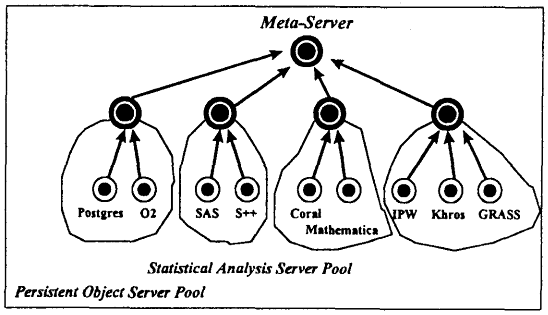
The server-pool boundary is then effectively-defined in terms of the content of registration maintained by a particular meta-server.

The idea of meta-server has been used in many systems. In the UNIX/SUN environment, inetd and yellow-page daemon are such examples. In the context of database systems, however, we are not aware of any system that implements the notion of a meta-server, although its architecture may employ a client-server model between a query engine front-end and a storage manager back-end.

The role of meta-server is not limited to flat server pool management. We can impose hierarchies and organize servers and the server pool in arbitrary ways. ⟨Figure 4⟩ depicts one such example where servers are organized in sub-pools based on their functionality, i.e., whether they are for image processing, statistical analysis, etc. We make two observations here that all servers are the client of at least one of the meta-servers, and that all servers may be the client of any number of servers in the pool.



⟨Figure 3⟩ Server Pool Management by Meta-Server


Meta-Server
Postgres
O2
SAS
S++
Coral
Mathematica
IPW
Khros
GRASS
Statistical Analysis Server Pool
Persistent Object Server Pool

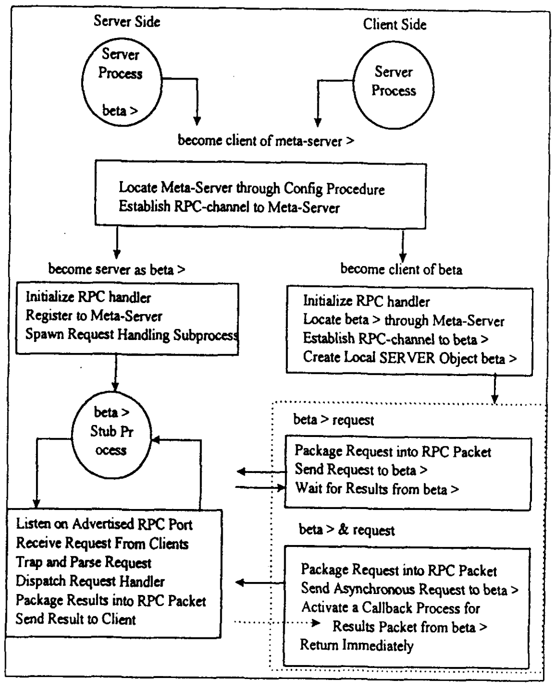
⟨Figure 4⟩ Hierachical Organization Server Pool

2. Meta-Server Methods and Configuration

Some of the major methods associated with a meta-server include list and find : The list method will return the set of all server names registered at the time of the method invocation. The find method will return the port address of the server given as an argument.

Via meta-server, clients can locate any server currently available. One of the remaining problems is that how the clients locate the meta-server itself? A common solution to this problem involves the notion of "well-known-port"; port; every client process has a hard-coded address of communication port, on which the meta-server will listen for the duration of its existence. We find this approach only partially satisfactory because it would unduly burden system maintenance and configuration. Cases may arise when the meta-server should be reconfigured to have different addresses (e.g., a particular host crashes).

Just like objects are abstracted and referred to by its Oid, the implementation details of the meta-server (e.g., its host and port address) should be abstracted and be made transparent to clients. The OMEGA meta-server will have its configuration information encapsulated in a procedure, which is shared among the meta-server and other servers and clients. At the time of initialization, meta-server will invoke this procedure to position itself in the network host/port address space dictated by the configuration procedure. Client processes, then, invoke the same procedure to figure out the address of the meta-server. The procedural configuration allows the system administrator to freely change the location of the meta-server by simply changing the configuration procedures code. This idea is similar to the procedural encapsulation of 'method' interface for objects and their behaviors.


Server Side
Client Side
Server Process beta >
Server Process
become client of meta-server >
Locate Meta-Server through Config Procedure
Establish RPC-channel to Meta-Server
become server as beta >
become client of beta
Initialize RPC handler
Register to Meta-Server
Spawn Request Handling Subprocess
Initialize RPC handler
Locate beta > through Meta-Server
Establish RPC-channel to beta >
Create Local SERVER Object beta >
beta > Stub Process
beta > request
Package Request into RPC Packet
Send Request to beta >
Wait for Results from beta >
Listen on Advertised RPC Port
Receive Request From Clients
Trap and Parse Request
Dispatch Request Handler
Package Results into RPC Packet
Send Result to Client
beta > & request
Package Request into RPC Packet
Send Asynchronous Request to beta >
Activate a Callback Process for
Results Packet from beta >
Return Immediately

〈Figure 5〉 OMEGA Client-Server Protocol

V. Protocols and User Language Constructs for Server Management

In this section, we present user-level language constructs for creating and interacting with server objects. The overall protocols for client-server interactions are summarized in 〈Figure 5〉, which will be elaborated in the following sections.

1. Becoming a Server

Any process can make itself available as a server by invoking a simple command '**become_server_as**' followed by the name by which the current process wants to be known to clients. Registering itself to a meta-server will be taken care of internally. The underscores are optional, e.g., '**become server as**' will be parsed exactly the same. If this command is invoked interactively, the session prompt will be changed from the default '**OMEGA>**' to the user-specified server name. When the current process (as a server advertised as '**alpha>**') is accessed from remote clients, only the routines defined in terms of trap will be made available to the public as the services of **alpha>**

```
OMEGA> become server as alpha>
alpha>
```

The command may have optional arguments for specifying the desired port number(by default, the system will select one from available ports). This command triggers the OMEGA parser to execute the following logical steps:

① spawn a subprocess to handle requests from the clients
② initialize a RPC-based port and trap routines
③ locate the meta-server, 'meta〉', and register to it a name and a network port
④ have the subprocess listen on the advertised port for any requests from the clients
⑤ trap and dispatch any subsequent client request

2. Writing Trap Routines

The trap routines intercept client requests and translate them into procedures native to the server. SOUL provides a directive called 'trap', and writing trap routines is very similar to writing ordinary method procedures using the programming constructs of SOUL. We will look at an example of a trap routine for the GRASS package. GRASS provides '**m.ll2u**' for converting a map of the Lat/Long coordinate system to a new map of the UTM system. In OMEGA, we may use this facility of GRASS as '**geog2utm**' methods available as a service from the server 'grass〉'. In this case, the necessary translation is very simple as is shown below.

```
trap geog2utm args {
        m.ll2u $args
}
```

3. Becoming a Client

In the following sequence of examples, we show how a user may (1) find out which servers are currently available, (2) pick a particular server and become its client, and (3) submit requests that the server is known to handle.

```
OMEGA> meta-server> list
==> grass> mathematica> coral> . .
OMEGA> become client of grass>
==> SERVER object "grass>" is created.
OMEGA> grass> d.rast soilmap
```

Any process can make itself a client to servers by invoking a simple command '**become client of**' followed by the name by which the server process is registered to a given server pool. Note that thescoping of '**grass>**' is local to the calling process (client). More specifically, this session has created its local link to the server '**grass**'; it becomes **grass**'s client.

4. Asynchronous Server Interface Protocols

By default, all the requests to, and replies from the server are synchronous. There are, however, cases where the synchronous communication is undesirable. For example, when an execution of a request takes a long time, clients may not want to wait until the entire processing finish on the server side. A more drastic situation may be when the server processes crashes, clients would unknowingly wait for replies that will never be delivered.

OMEGA supports the asynchronous mode of interface for SERVER. Users may choose the mode of interface at each time of method invocations. If the next task of a client is not dependent on the reply from the server, the client may submit request and immediately resume its local tasks. The server side, after finishing the request, will notify the client and deliver the result asynchronously. This mode is similar to the handling of background jobs in the UNIX environment. SOUL is augmented by the constructs for the asynchronous **SERVER** interface. If the server object name is suffixed by the symbol '&', then the method request is sent to the server in the asynchronous mode. The following example shows that the client may resume other tasks right after it submits an asynchronous request.

```
OMEGA> grass>& d.rast soil
OMEGA>
```

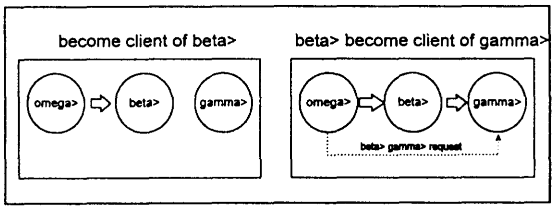
'&' is chosen for the asynchronous request specification since it is commonly understood as a 'background' job request in the UNIX environment. When the server side finishes processing the request, it notifies the client by beeping on the client's terminal, for example. It then writes the results in a location (named by a variable) so that the client can fetch the results later. This protocol is implemented by way of 'callback' facilities. Callback is a routine which a client piggybacks along with a request. The server will invoke the callback routine with the results as a parameter.

5. Indirect Access to Server

We now describe an indirect accesses to SERVER objects and the situations that call for such interfaces. Consider, for example, an application that requires services from two hypothetical servers, **beta>** and **gamma>**. Further consider that **gamma>** is separated (for reasons such as network 'firewall') from the current process. If **beta>** can access **gamma>**, then we might be able to ask beta> to pass our request to **gamma>** and also deliver the results from back to us. In this case, **beta>** plays a role of a middle 'agent' who relays requests and results between us and **gamma>**. This access mode to a server through agents is called indirect access.

There will be another situation where indirect accesses make sense. Suppose we need to process a large dataset (e.g., a Landsat Image), which is stored and managed by **beta>**, **gamma>** is an image processing package that runs the particular algorithm that we need. It would be very inefficient, in this case, if we first retrieve the dataset from **beta**, and then pass this dataset as a parameter to **gamma>** for processing; large dataset dataset has to be moved twice over the network. Instead, we would like to have **beta>** send its dataset directly to **gamma>**, and relay to us just the results. This indirect access involves just a single transfer of the dataset.

The required steps to set up the indirect access of our example will be as follows. First, the current process that runs our session of OMEGA must be made a client of **'beta'**. Second, a request is sent to **beta**, asking **beta>** to **'become a client of gamma>'**. Now the servers are correctly configured and we are ready to access **gamma>** indirectly for its service **'gamma's_expertise'** by asking **beta>** to make the request to **gamma>**.



〈Figure 6〉 Indirect Access to **gamma〉** via **beta〉**

Ⅵ. Implementation of OMEGA Prototype

The architecture of the OMEGA system is guided primarily by the goals of extensibility and loose-coupling of interoperable subsystems over the network. OMEGA can be dynamically configured to respond to the varied requirements of new and different applications by integrating either public domain or off-the-shelf commercial packages. The OMEGA infrastructure consists of several layers of system components which interact in the mode of client-server and can always be replaced with new, experimental components. A fully operational version of the OMEGA prototype has been completed and released through WWW. The OMEGA shell runs on UNIX platforms, and porting to other platforms platforms is currently underway. It consists of some 15,000 lines of Tcl as well as 2,000 lines of C programs on top of a number of libraries. The important components include an extensible OMEGA kernel of class library, a SOUL parser, a graphical user interface, a persistent storage server, computation server coupling module, and a gateway server as shown in 〈Figure 7〉. We describe some of the components in the following sections.

1. Data Server

The data server component comprises a database back-end that manages the physical storage of data items. OMEGA currently employs the storage manager subsystems of POSTGRESS (Stonebraker and Rowe, 1986) as a platform for persistent data server. The OMEGA data server provides persistency to valid object, be it an object, a class object, or a method object. The persistent store is shared and accessed by multi-

```
OMEGA> become client of beta>

OMEGA> beta> become client of gamma>

OMEGA> beta> gamma> gamma's_expertise
```

users. OMEGA relies on the Postgres transaction mechanisms (based on 2PL-protocol) to control concurrent accesses. In OMEGA, the Postgres storage manager runs behind the scene. The user interacts with Postgres indirectly through the primitives, such as **persist**, **load**, and **perish**.

A direct interface to Postgres is also possible within OMEGA in terms of the local **SERVER** object named **postgres>**. The non-OMEGA data stored in Postgres can be accessed by redirecting POSTQUEL to the server.

2. Computation Server

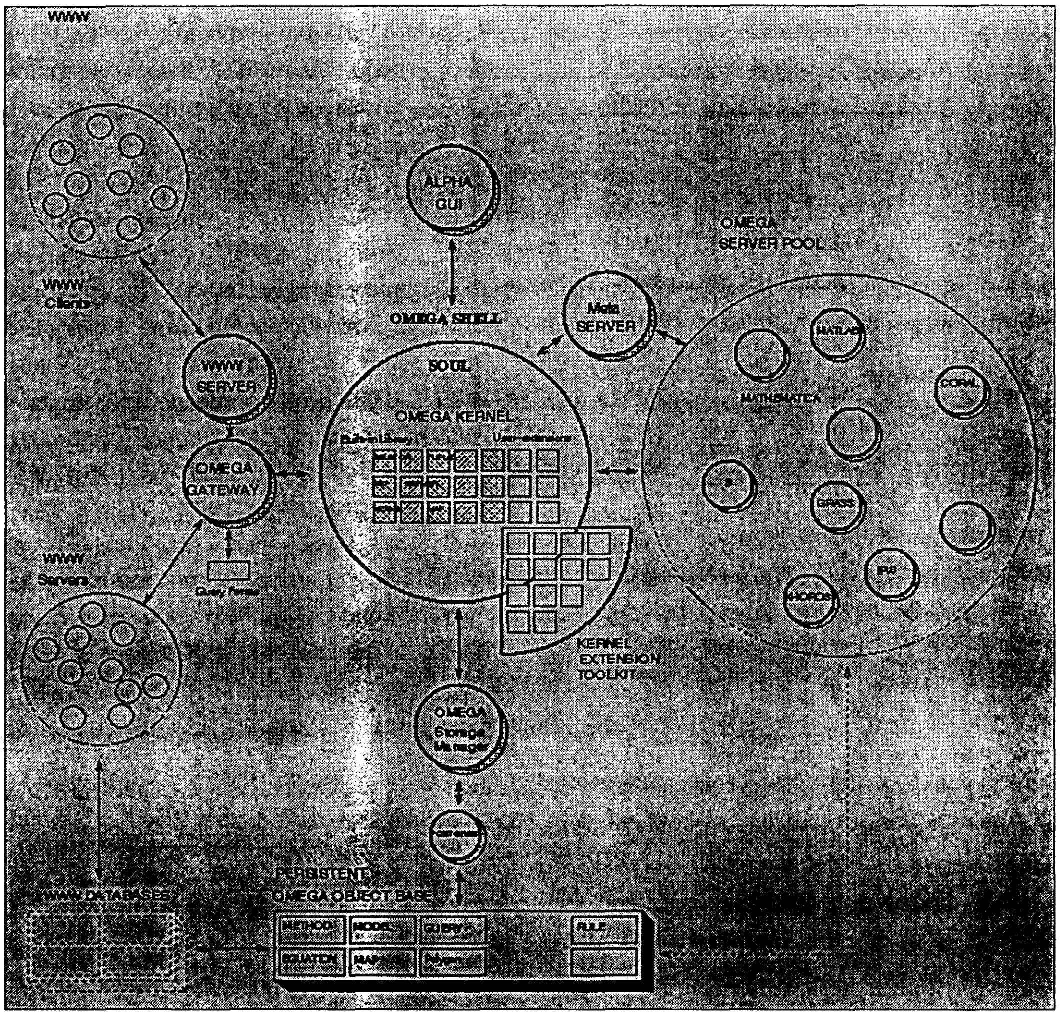
OMEGA is a comprehensive application development environment consisting of numerous servers for highly specialized computations. We illustrate some of the integral subsystems coupled in OMEGA according to their functionalities.

1) Equational Object Server

OMEGA employs **'mathematica'** as a specialized computation server for a wide range of mathematical calculations. Equations and other mathematical structures are represented in symbolic expressions. Manipulation of algebraic formulas is well supported as are differential equations and other numerical computations such as integrations and linear programming. In the OMEGA environment users will call upon mathematical computations through a server object 'math'.

2) Axiomatic Rule Object and Inference Server

OMEGA employs CORAL as an inference engine to process axiomatic rule base. CORAL is a logic-based database language developed at the University of Wisconsin-Madison(Ramakrishman et al., 1993). It seeks to combine features of a database query language with those of a logic programming language. It is distinguished from conventional database languages by its inference capabilities based on first-order logic. Although its syntax is not entirely suitable for general purpose programming, the rule-based inferences are attractive because they represent and process certain knowledge succinctly and declaratively. We have used CORAL in implementing an early pilot system of OMEGA. OMEGA entrusts CORAL with such tasks as rapid prototyping of algorithms, algorithms, constraint programming, and problem-



⟨Figure 7⟩ System Architecture of OMEGA

solving based on transitive closure. In the OMEGA environment, users will call upon CORAL's inference capabilities through a server object 'coral'.

3) Image Processing Subsystem

OMEGA employs **'Khoros'** and **'IPW'** for their image processing capabilities. IPW (image processing workbench) is a development environment for image manipulation algorithms and applications with particular emphasis on satellite imagery(Frew, 1990). IPW offers a single, portable image data format that accommodates both integer and floating-point data representations, as well as an unlimited amount of ancillary information. Image data may have an arbitrary number of channels or bands. In the OMEGA environment, users will call upon image processing operations of IPW through a server object **'ipw>'**. Users can also use SOUL scripting for programming the sequence of desired actions.

4) GIS Subsystem

The OMEGA environment provides the core functionality of a GIS as its sub-system. GRASS (CERL, 1993) is imported into OMEGA as a spatial operator library, which is made available either through the server object **'grass>'**, or through the set of OMEGA primitives trapped into GRASS procedures. The OMEGA shell provides a powerful front-end to GRASS, in which the subroutines native to GRASS may intermix with other programs to produce arbitrarily complex applications. The data files on which GRASS programs operate are encapsulated as OMEGA objects, and the GRASS programs are encapsulated as methods. The OMEGA persistent store provides a reliable database back-end to GRASS. 〈Table 1〉 lists examples of the GRASS programs that are 'wrapped' as OMEGA's primitive spatial operators.

Table 1. GRASS Program Conversion

OMEGA		GRASS Program
class	method	
Map	geog2utm	m.ll2u
Map	utm2geog	m.u2ll
Vector	vect2rast	v.to.rast
Raster	info	r.info
Raster	theme	r.cats
Raster	overlay	r.cross
Raster	combine	r.combine
Raster	filter	r.infer
DEM	basin	r.basins.fill
DEM	drain	r.drain

Ⅶ. OMEGA Gateway Server

Information access through a wide-area network is exploding at a remarkable rate. The existing hardware and software to tap on the Internet through World Wide Web(WWW) has matured. WWW is an embodiment of the universe of network-accessible information. With the advent of WWW technologies, the possibility has arisen to provide information holdings of OMEGA to much wider clients. The main thrust of the OMEGA gateway server is that we cannot expect every remote site to have an OMEGA system installed, but we can still expect them to access resources maintained by a local OMEGA through WWW, and vice versa. The WWW-based access to data in OMEGA is 'self-sufficient' since the of the data are also made available. The subsystem of OMEGA for exchanging information across networks is founded on the so-called 'gateway' technology. We have configured and implemented an operational gateway server of OMEGA. This section describes the motivation, and sketches architecture and some of the engineering details of our implementation describes the OMEGA gateway servers in greater detail.

1. Why Gateway?

The success of collaborative scientific researches relies on effective and efficient information exchanges among distributed sites. The information exchanges involve datasets, programs, models, and analysis results. The datasets fetched (through ftp) would be of little use without the processing modules to manipulate them. The current practice of data exchange involves importing raw data from remote sites, and developing programs locally to process them. If the remote sites also maintain processing programs, they will be fetched and compiled locally. In any case, there will be traffic of either moving data to algorithms, or moving algorithms to data. The self-sufficient mode of information exchange, however, will only involve migrations of 'value-added' data being processed at the site of the information provider. The 'Archie' and 'Gopher' servers have some capabilities of queries processing to filter desired datasets. But these processings are limited to simple search conditions. The general mechanisms for data transformation and manipulation are absent in these servers. The 'gateway' referes to a piece of software coupled with a WWW server. Gateway servers are executable programs that can run by themselves. They have been made external programs in order to allow them to run under various information servers interchangeably. A gateway server accepts queries from remote clients through a locally installed HTTP-server. The queries will be processed in the local query engine, and the results will be sent back to the clients. The results are usually packaged by HTML directives. These directives tell the client (such as Netscape) how the data should be interpreted and possibly visualized. The main motivation of incorporating gateway technology into OMEGA is to serve and exchange information resources maintained by OMEGA with other information sources. Simply put, the benefit of the OMEGA gateway is that not only the data holdings but also the entire functionalities of OMEGA are made available to remote users. OMEGA gateways offer the following benefit in distributed object management:

① It allows self-sufficient interface to clients from remote, external, and heterogeneous environments.
② It guides remote clients via query-forms which are adaptive to database states.
③ It servers as network-wide data exchange port.
④ Remote systems (i.e., their capabilities) are 'connected to' through gateway rather than 'installed' as a local copy.

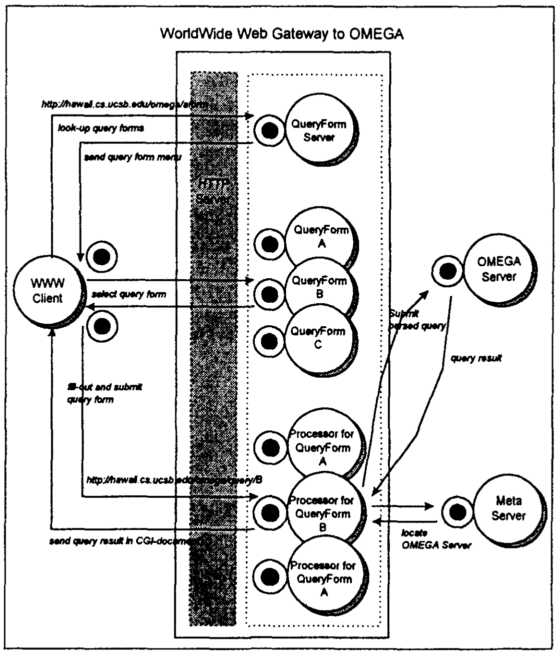
2. Construction of OMEGA Gateway Server

A particular standard of a gateway interface protocol gaining wide acceptance among the WWW is 'Common Gateway Interface' (CGI). A WWW-server that is configured to accept CGI protocol has been developed in National

Center for Supercomputing Applications (NCSA): http (current version 1.1). The OMEGA gateway server is implemented to meet the specifications of the CGI protocol and employs RPC as the backend which handles WWW clients and their requests in CGI. The OMEGA gateway is a stateless server in the sense that no state information is cached internally by the server side on behalf of the client. The necessary information is piggy-backed to and from clients in the form of a URL (Universal Resource Locator). The implementation and configuration of the OMEGA gateway server and a WWW front-end server involves the following steps:

① local installation of 'httpd' with an URL
② external view and access configuration of the OMEGA system including a gateway query form server
③ compilation of CGI-conforming query forms
④ installation of a gateway query form parser
⑤ coupling of gateway servers with the meta-server and the OMEGA kernel

The gateway query form is a mechanism for soliciting the information from a remote user. It consists of the dialog window. A variety of graphical interface elements are embedded within the query form to which users can point and click to fill out the query request form. The interface elements typically ranges over the menubar, scrollable selection windows, buttons, text input dialog boxes, etc. The idea of multiple query forms and a query form server can be described in the following interface scenario. First, on the client's request, the query server suggests a set of query forms accepted by the server. Then the client decides the format in which he will express his queries and fills out the query form and submits it back to the query server. The query server will pre-parse the submitted query, translate it into SOUL, have the local OMEGA server process the query, and finally, deliver the packaged results back to the client. This protocol is depicted in ⟨Figure 8⟩.


WorldWide Web Gateway to OMEGA
http://hawaii.cs.ucsb.edu/omega/aforms
look-up query forms
send query form menu
HTTP Server
QueryForm Server
QueryForm A
QueryForm B
QueryForm C
WWW Client
select query form
fill-out and submit query form
http://hawaii.cs.ucsb.edu/omega/query/B
send query result in CGI-document
Processor for QueryForm A
Processor for QueryForm B
Processor for QueryForm A
OMEGA Server
Submit parsed query
query result
Meta Server
locate OMEGA Server

⟨Figure 8⟩ WWW-Gateway Interface of OMEGA

Ⅷ. Conclusion

A distributed and heterogeneous computing environment is the rule rather than the exception in experimental geoscientific research conducted on high-level testbeds consisting of unique hardware and software architectures. The ensuing problems in accommodating heterogeneity and integrating various components have been widely recognized. OMEGA is an environment where heterogeneous computer systems share a small set of key interface primitives. We emphasize that the OMEGA environment is a loose form of interconnection at the service level, rather than seeking to construct a transparent operating system bridge between heterogeneous systems. In other words, OMEGA accommodates multiple standards and the autonomy of individual systems (instead of legislating another standard) while the integration is still accomplished among a large number of system types with a small number of instances. We are convinced that loose-coupling is the only viable design to meet the highly dynamic nature of geoscientific modeling for its demand of various tools.

It is the **SERVER** class that provides the necessary software infrastructure to minimize the 'hacks' of adding new systems to an existing environment. We developed a system abstraction of **SERVER** and associated user level primitves, and implementation techniques that bridge theory and practice. We investigated the extent to which abstraction techniques and language primitives can hide low-level details about data and computations involved in advanced geoscientific applications. An initial evaluation of our work based on a prototype implementation of OMEGA is very positive and encouraging.

References

Asrar G. and D. Dokken, 1993, *EOS Reference Handbook*, NASA Publication: NP-202, March 1993.

CERL, 1993, *GRASS 4.1 User's Reference Manual*, Construction Engineering Research Lab.

Frew, J., 1990, *The Image Processing Workbench*, PhD thesis, University of California, Santa Barbara, Santa Barbara, CA., July 1990.

Object Management Group, 1997, *Common Object Request Broker: Architecture and Specification*, Technical Report.

Park, K., 1994, SOUL of OMEGA: *Design and Implementation of an Object-based Modelling Environment for Geoscientific Applications.*, PhD thesis, University of California, Santa Barbara, Santa Barbara, CA., June 1994.

Ramakrishman, R., Seshadri, P., Srivastava, D. and Sudarshan, S., 1993, *The CORAL Manual*, Computer Science Dept., Univ. of Wisconsin-Madison, Madison, WI.

Schek, H. and A. Wolf, 1993, "From Extensible Databases To Interoperability Between Multiple Databases and GIS Applications", in *Proc. of 3rd Int. Symposium on Advances in Spatial Databases*, (Singapore), pp. 207-238, Springer-Verlag.

Smith, T., Su, J., Abbadi, A.El, Agrawal, D. and Saran, A., 1995, "*Computational Modelling Systems*," *Journal of Information Systems*, vol.19, no.4.

Stonebraker, M. and L. Rowe, 1986, "The Design of POSTGRESS," in *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, (Washington, D.C.), pp 340-355, May 1986.