From Metacomputing to Metabusiness Processing*

Li-jie Jin
ljjin@hpl.hp.com
Software Technology Laboratory
Hewlett-Packard Labs

Andrew Grimshaw
grimshaw@cs.virginia.edu
Computer Science Department
University of Virginia

Abstract

The importance of large-scale electrical business processing is increasing today as recent Internet technologies build on the basic infrastructure. Simply integrating existing technologies and resources to form a platform that satisfies large-scale electrical business processing requirements is not enough, however. Legion is a wide-area distributed object system that offers mechanisms for describing, creating, and managing objects in a large-scale, heterogeneous, distributed computing environment. Its original design objective was to build a global virtual-computer system that uses Legion as its operating system for compute-intensive applications. This paper introduces our efforts to extend the Legion system into a backbone that supports business processing with consistent resource representations, identical service interfaces, and an easy-to-use developing environment. We will focus on a framework that supports CORBA from within the Legion system.

Keywords: metabusiness processing, Legion, CORBA

1. Introduction

1.1 Business processing and CORBA

E-commerce has grown dramatically in recent years. Business enterprises need effective, flexible, scalable and reliable information platforms in order to adapt to changing market and global competition. Information platforms must cooperate in order to handle the rapidly increasing number of business-to-business electric transactions. This in turn requires a very large business processing system that may handle thousands of sites, tens of thousands of users, and hundreds of thousands of processes [8].

Workflow management technology [14] [18], developed in the last few years, can handle reengineering business and information processes, business process automation and application integration. It supports the everyday operation of many enterprises and their work environments. But to effectively support workflow management, businesses must adapt their existing computing environment to a new distributed component-oriented environment [4]. This new environment should support evolution, replacement, and new workflow applications or component systems as processes are reengineered. It should also have consistent resource representations, identical service interfaces, and easy-to-use developing support.

Many commercial Workflow Management Systems (WFMS) are well suited for simple form-based office procedures [2]. However, more complex applications (such as production process control, telecommunication service provisioning, insurance

* This work partially supported by DARPA (Navy) contract # N66001-96-C-8527, DOE grant DE-FG02-96ER25290, DOE contract Sandia LD-9391, Northrup-Grumman (for the DoD HPCMOD/PET program), DOE D4590 00-16-3C and DARPA (GA) SC H607305A

Report Documentation Page		*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE **2000**	2. REPORT TYPE	3. DATES COVERED **00-00-2000 to 00-00-2000**
4. TITLE AND SUBTITLE **From Metacomputig to Metabusiness Processing**		5a. CONTRACT NUMBER
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)		5d. PROJECT NUMBER
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of Virginia,Department of Computer Science,151 Engineer's Way,Cahrlottesville,VA,22094-4740**		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSOR/MONITOR'S ACRONYM(S)
		11. SPONSOR/MONITOR'S REPORT NUMBER(S)
12. DISTRIBUTION/AVAILABILITY STATEMENT **Approved for public release; distribution unlimited**		
13. SUPPLEMENTARY NOTES		
14. ABSTRACT		
15. SUBJECT TERMS		

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES **10**	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**			

claim handling, and medical services) need high performance, scalability, reliability, automatically enforced consistency, etc. Existing commercial WFMS do not scale well, have limited fault-tolerance, and are inflexible when interoperating with other workflow systems [1]. WFMS developers consider the Common Request Broker Architecture (CORBA) [Siegel 96] and other technologies, including DCOM and Enterprise Java Beans, to be solutions to these problems.

CORBA is a standardized specification for a distributed-object platform architecture. It aims to archive application interoperability in multi-vendor networked environments. It has many attractive features that match the requirements of large-scale business processing. For example, the OMG (the Object Management Group) IDL is an interface definition language used to define CORBA object types by defining their interfaces. With IDL-described interfaces, CORBA objects are able to interoperate with each other in a transparent way and in a fully distributed environment. The mappings between OMG IDL and other languages (such as C, C++, Smalltalk and Java) are included in CORBA specifications. The General Inter-ORB Protocol presents WFMS the possibility to construct interoperable data-support infrastructures. CORBA also offers many object services that support reliable and correct execution of business processes. In early 1997, OMG set up a Workflow workgroup to establish a Workflow Management Facility (WfMF) specification. Researchers at Newcastle University, University of Georgia, Sema Group (Spain) and many other institutions and companies are using CORBA as the data supporting system of their workflow management systems [5] [21] [13].

However, CORBA is an architecture specification, not an implemented system. Most CORBA products comply with specifications but actual implementations differ from each other. They also implement different services in different manners.

These factors suggest that solutions which address distributed platform problems for large-scale business processing need to go beyond the CORBA specification.

1.2 Metacomputing systems

If an applications is to take advantage of the increasing availability of high-performance networks and processors, it must be able to share resources spread over complex, large-scale, heterogeneous, distributed environments spanning multiple administrative domains. This kind of environment is called a *metacomputing system*, a networked virtual supercomputer. Metacomputing systems share many characteristics with traditional parallel and distributed computing systems. Like distributed systems, they collect geographically separate resources of varying capabilities through high-performance but sometimes unreliable networks and share those resources among geographically separate users. Like parallel computing systems, they allow users to divide and schedule their application tasks carefully in order to balance the communication cost and the usage of processing power. However, a metacomputing system distinguishes itself from parallel and distributed computing systems via its single, coherent, virtual system image. It is the metacomputing system's responsibility to support the illusion of a single machine by transparently managing data movement, caching, and conversion; detecting and managing faults; ensuring that the user's data and physical resources are adequately protected; and scheduling application components on the resources available to the user [12]. A metacomputing system is designed to be a computing platform that can scale to a large-scale system, handle heterogeneity at multiple levels, allow effective interoperations among different system components, remain secure, and combine resources from different administrative domains. The high-performance application categories targeted by these metacomputing systems are: desktop supercomputing, smart instruments, collaborative environments and distributed supercomputing. Most of those applications that are running on a metacomputing system are computing-intensive; for instance, multi-scale climate modeling, computational fluid dynamics, real-time image processing and visualization.

We believe that a good metacomputing system can address these issues for large-scale business processing. We use the term *metabusiness processing* to describe business processing on metacomputing systems. Legion and Globus are two active metacomputing research projects [6] [3] that can support metabusiness processing. These two projects share a common base of target environments, technical objectives, and a number of similar design features. This paper will focus on Legion, however, and discuss our efforts to extend Legion's abilities so that it can support metabusiness processing. We first describe the Legion system's basic structure in Section 2. In Section 3, we lay out a framework to support CORBA specifications in Legion. In section 4, we discuss mapping object services between Legion and CORBA. We conclude our works by identifying topics for further research in section 5.

2. The Legion system

Legion is an object-based metacomputing system. Legion's major design objectives is to offer high-performance application programmers a powerful tool set and an easy-to-handle environment. This environment is in a heterogeneous, large-scale distributed system in which computing and storage resources belong to multiple organizations. Legion addresses distributed computing issues such as communication, task scheduling, resource management, security, fault tolerance, scalability and heterogeneity management.

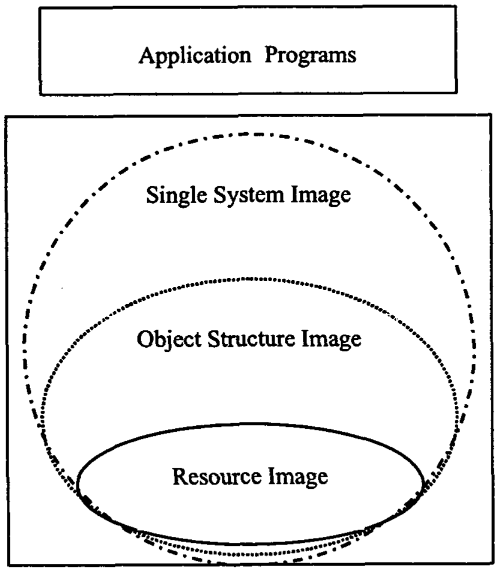

Figure 1. Legion images

We can understand Legion from three different viewpoints: Legion's resource image, Legion's object structure image, and Legion's single system image. These images' relationships are shown in **figure 1**.

2.1 Resource image

Resource management is a typical operating system problem. As a wide-area operating system, Legion utilizes various resources that may be distributed across the Internet. These resources can include processors, memory, network, persistent storage, I/O devices, databases and application tools and environments and may be owned and controlled by different organizations. Legion provides a consistent interface to share and manage these resources over networks. Inevitably, this diverse collection of resources involves different architectures, processors, data representation formats, data alignments, system configurations, and operating systems. This makes a challenging environment for a would-be application programmer.

Legion looks at resources as physical components that together form a worldwide virtual computer system (**figure 2**). It represents and manages these components with objects that share identical operating interfaces and various attributes. The objectives of resource management in Legion are: (1) to reduce completion time of certain tasks through parallel execution; (2) to share spare or expensive resources between users who may be separated by large distances; and (3) to support collaborative works by collecting resources from different administrative domains into a single application working environment. In many ways, Legion's resource management objectives and strategies are similar to WFMS, except that no human resources and activities are currently involved in Legion resource management.

For example, the Harvard Medical School is performing research on the causes and symptoms of multiple sclerosis. The core research group has developed image-processing pipelines that build three-dimensional models of characteristic brain lesions from MRI scans. To significantly advance the research, they need MRI scans from multiple partner institutions as well as a database of their own image-processed results for their research partners. As a first step, they would like a tool that can automatically identify MRI scans pertaining to the study whenever they are made at partner hospitals, securely move those scans over the Internet to Harvard, and then process them. Very little administrative support for the tool can be expected at any of the partners. More metacomputing application cases can be found in [6].

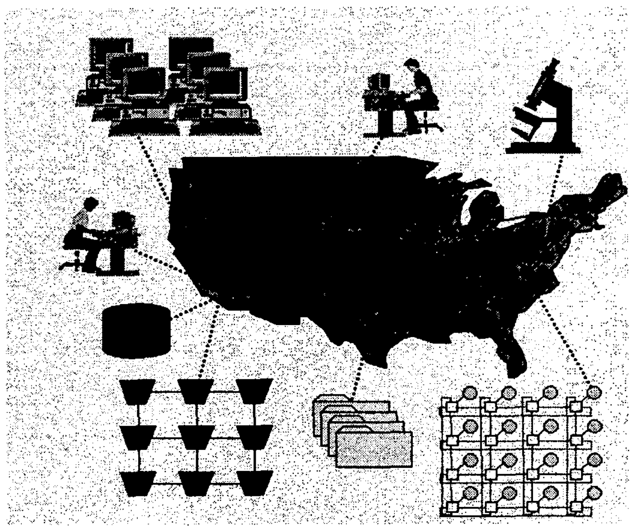

Figure 2. Resources of the Legion system.

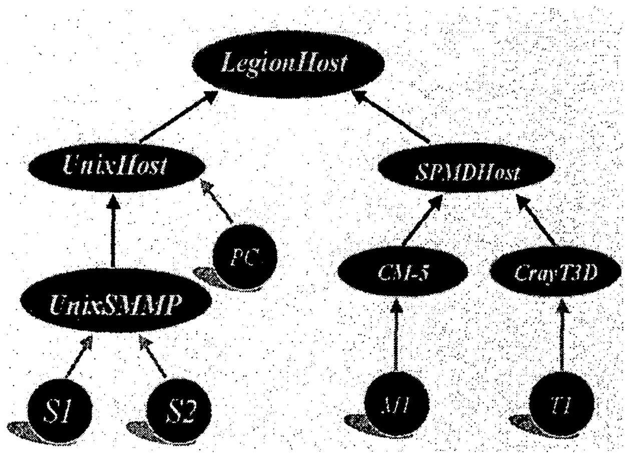

Figure 3. Legion class derivation

2.2 Legion's object structure image

In Legion's object model [11], each element, including all kinds of resources, is represented by an independent, active object. For instance, processors are represented by host objects, storage devices are represented by vault objects. Legion objects are all instances of classes and each class is itself a Legion object, a metaclass object. Class objects are responsible for creating and managing their instances and for selecting appropriate security and object placement policies in a distributed environment. These objects communicate by means of Legion's remote invocation service. All Legion objects export a common set of object-mandatory member functions, such as deactivate(), getInterface(), and add_attributes(). Class objects also export a set of class-mandatory member functions, such as createInstance(), activateInstance(), and deactivateInstance().

Legion defines the interfaces and functionality of several core class objects, such as LegionClass, LegionHost, LegionVault, and LegionBindingAgent. LegionClass is the base class of all other Legion classes and provides the full set of Legion's class-mandatory member functions The core Legion classes provide mechanisms for the user-level classes to implement appropriate policies and algorithms. They set the minimal interface that the core objects should export. LegionHost, LegionVault, and LegionBindingAgent are base classes for Legion's core class types, including host objects, vault objects, and binding agents. Legion allows users to alter class objects so that they can apply their own object management strategies to their physical resources.

Figure 3 demonstrates how processing resources are organized and managed by Legion objects and the relationship between those Legion objects. Vault objects and binding agents have similar class hierarchies.

2.3 Legion's single system image

The single system image in the Legion system means that Legion provides a universal shared name space that names all objects of interest to the system and its users. These objects represent files, processes, processors, storage, users, and services. This allows any Legion object to transparently access any other Legion object without regard for location or replication. Single system image also means that, when a user object is created Legion should allocate processor and disk space for this object transparently if user does not intend to explicitly place the object on a particular host or disk.

For example, users who login to the same Legion system can use legion_ls, a Legion command-line tool, to see the same Legion context space information of system components, tools and user applications (**figure 4**).

```
$ legion_ls -R
.                          (context)
class                      (context)
etc                        (context)
home                       (context)
hosts                      (context)
impls                      (context)
vaults                     (context)

In context "class":
.                          (context)
..                         (context)
AuthenticationObjectClass  (class)
BasicFileClass             (class)
BasicSchedulerClass        (class)
BindingAgentClass          (class)
ContextClass               (class)
LegionClass             (legion class)
UnixImplementationClass    (class)
```

Figure 4. Output of Legion command legion_ls

Given that the Legion system is running on top of a large-scale, distributed, heterogeneous computing platform, the Legion single system image masks a great deal of complexity from the users' view.

3. A framework to support CORBA inside the Legion system

CORBA is a distributed object architecture specification proposed by a consortium of around eight hundred partners. Its basic objective is to achieve interoperability between applications that reside in a heterogeneous computing environment. By supporting CORBA, Legion and participating applications can cooperate with other CORBA systems and CORBA applications via standard interfaces. It is the first step towards a metabusiness processing system.

3.1 Legion and CORBA

Legion and CORBA have different design objectives. Legion focuses on sharing computing resources on a wide-area network and on presenting a single virtual computer image to application users, as well as hiding resource heterogeneity from application programmers. Legion integrates considerations for scalability, security and fault tolerance issues in a wide-area computing environment into each level of its design and implementation. Legion adopts a macro dataflow execution model.

The CORBA specification, on the other hand, concentrates on software integration issues. It emphasizes interoperability of components through standard interfaces. CORBA has a client/server execution model. Many CORBA products comply with specifications, but actual implementations differ in the way they implement services.

Both, however, use object-based architectures and have interfaces that could be described with some kind of IDL. Legion and CORBA support component interoperability between multiple programming languages and heterogeneous execution platforms. They also both use object wrappers to support legacy code. These common features are the foundation for designing a framework in Legion to support CORBA standard.

3.2 How to make Legion CORBA-compliant

A CORBA-compliant system should include an Object Request Broker (ORB) with a Basic Object Adapter (BOA). It should have an IDL compiler(s) that supports mapping between OMG IDL and at least one high level language, such as C++, C, or Java. We could design a new CORBA ORB sub-system for Legion. This approach would allow us to stick to the CORBA specification from the design stage. The problem is that there will be a lot of redundant work implementing

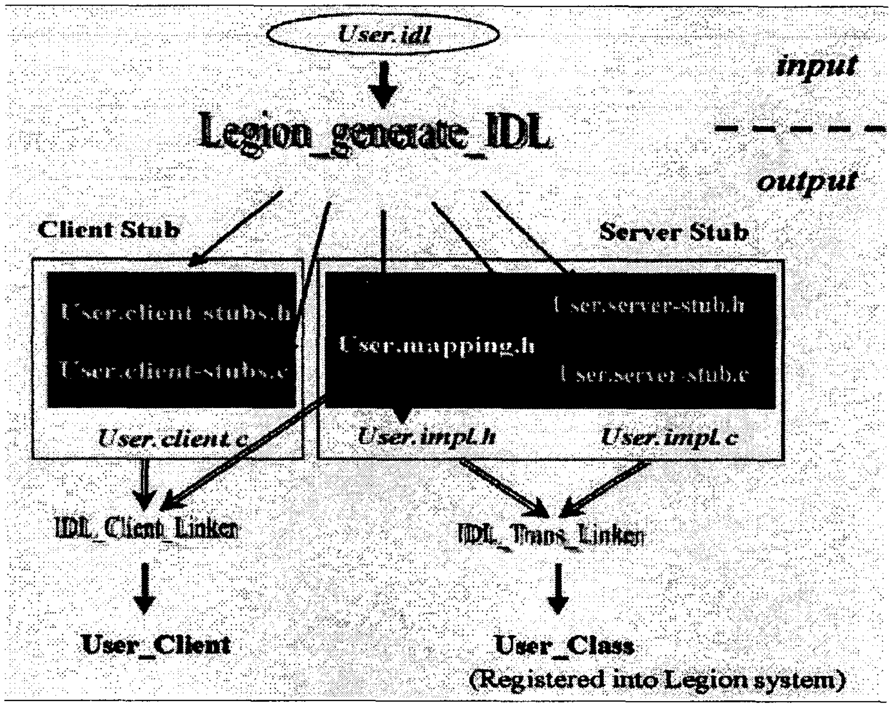

Figure 5. How to support CORBA applications in Legion

functions and services already in the Legion system but needing slightly differences in interface description and semantics.

Or, we could integrate Legion with another CORBA implementation, such as Orbix or Visibroker. The advantage of this strategy is that we only need to build some kind of bridge mechanism between Legion and other ORB system. The disadvantage of this strategy is that Legion is crippled when handling CORBA-related applications. It would also introduce two additional data-transfer layers for CORBA applications.

We could try designing a group of library functions to support CORBA applications. This is the easiest way to implement the supporting framework but it's hard to use. It's unlikely that we can ask CORBA users to call Legion functions in their implementation code when they want to do their work on a so-called CORBA-compliant platform.

Finally, we can use Legion's distributed object model and object services to simulate activities of a CORBA ORB and a BOA. We can then implement a compiler to perform the mapping between the OMG IDL and high-level programming languages, such as C++ and Java, and the mapping between CORBA's object services and Legion's object services. In this scenario Legion itself will work like an ORB. Legion cooperates with other ORBs through public interfaces described in the OMG IDL. CORBA applications can get direct support from the Legion runtime library without communication between redundant middle layers.

3.3 The CORBA supporting framework for Legion

Based on the discussion in section 3.2, **figure 5** shows a framework for supporting CORBA in the Legion system. Users who want to run CORBA applications on Legion should develop their server implementation code and client implementation code and have the public interfaces of their application described with the OMG IDL.

The Legion_generate_IDL is a C++/IDL compiler for the Legion system. This compiler takes the interface description, using the IDL as input, and generates five stub files that include all Legion-related method invocations. These stub files include object service mapping between CORBA object services and Legion object services. The application_name.client-stubs.h and application_name.client-stubs.c files are Legion client stub code, and will be compiled and linked with the client implementation code to form a executable CORBA/Legion client program. The application_name.server-stub.h, application_name.server-stub.c and application_name.mapping.h files will be compiled and linked with the server implementation code to form a CORBA/Legion server program. This server program will be registered into the Legion system before it is ready to accept requests from the client program. The application_name.mapping.h includes the implementation code of public interface mapping between the OMG IDL and C++. Legion offers tools such as legion_get_interface to allow applications or users to get interface information for any object in the Legion system.

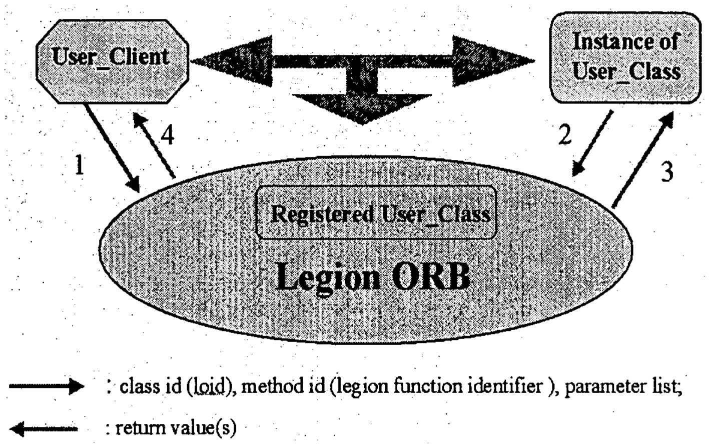

Figure 6. Execute a CORBA program in Legion

Figure 6 shows a top view of running a CORBA application on Legion. The object request issues from the client program (1). The client stub translates the request into Legion's object request format and submits the request to the Legion runtime library, Legion ORB. The Legion ORB locates the server object and sends a Legion object service request to an instance of the registered server program (here, the server program is User_Class) (3). The server stub translates the Legion object service request back to a format that the server implementation code can understand, i.e. the C++ format that is mapped from the IDL interface (2). The return value, if any, will follow the path backward from the instance of server object to the client program (4).

An Object Request Broker (ORB) should be able to locate and reference server objects. It should transparently mediate invocations on remote objects and their replies. It also conciliates diversities in machine and language formats by marshalling and unmarshalling request parameters and server replies. An ORB creates, activates, de-activates, and destroys instances of server objects using the operations of CORBA's life-cycle service or a replacement.

Figure 7 shows more detail of how the Legion runtime system works as an ORB when a CORBA application accesses its services through client and server stubs generated by Legion_generate_IDL. Object A is the client object and Object B is an instance of the server class object. In the client stub code, a Legion method handler is set up before the client issues any requests. After that, a Legion program graph [Viles97] is initialized. The input parameters are packed into a Legion buffer and added to the Legion program graph (1). The graph is then executed (2 & 3). At this point, the client program will wait for the return value from the Legion runtime system. If there is no exception, the return value will be available from a receiving Legion buffer (4). In server stub code, all available methods that the server object is going to export are registered into Legion's runtime system and are enabled to accept requests from clients. A parameter stack is set up to receive incoming parameters. Those parameters will be unpacked from a receiving Legion buffer and used to call the method exported by the server class object.

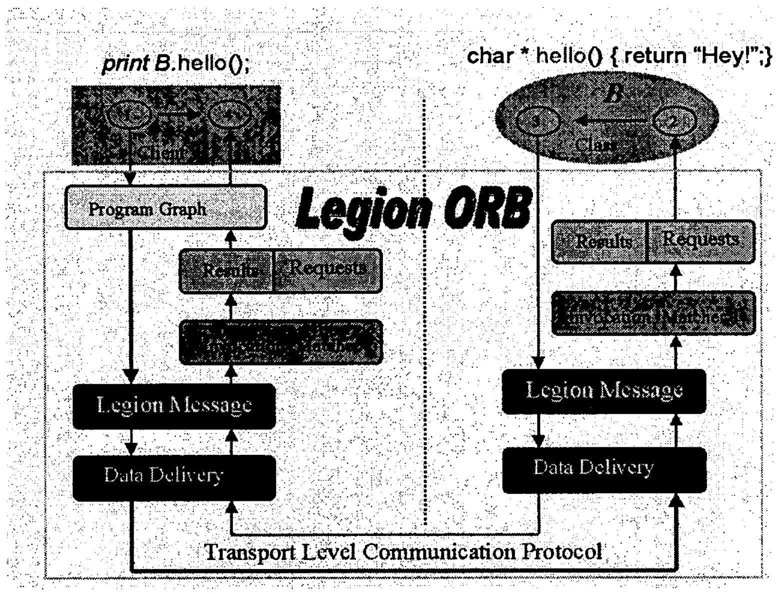

Figure 7. Working procedure of the Legion ORB.

The invocation matcher on the server side generates requests to an instance of the server class object when it receives all parameters of the invoked method. The invocation matcher at the client side matches the return values with the pending invocation and sends them to correct invocation.

4. Mapping Between Legion And CORBA

The object model and object services of CORBA and Legion are similar but have different implementation details and different interfaces. In CORBA's object model, objects are some encapsulated entities that perform services. An object is referred to by a unique object reference. The CORBA object model guarantees that a request invocation will provide a return. It may be either a valid return or an exception value. Based on this guarantee, client processing is largely simplified, since a client will not be left to hang up indefinitely because of failures from network, processing nodes, storage devices or server software. In Legion's object model, all Legion objects belong to classes and are logically independent, address-space-disjoint. Legion objects communicate via non-blocking method calls. They stay in one of two possible states: *active* or *inert*. Legion objects are named using also a unique LOID (Legion Object ID).

4.1 Life cycle service

The CORBA Life Cycle Service defines services and interfaces for creating, deleting, copying, and moving objects [17]. Life Cycle Service assumes CORBA implementations support object relocation. The "factory" is an object that creates another object. It also has well-defined IDL interfaces and implementations in some programming language. Factories provide the client with specialized operations to create and initialize new instances in a natural way for the implementation. Before creating an object in another location, the client should have information about a factory. Through contact with the factory object, the object is virtually created. To create an object, a client must posses an object reference for a factory, which may be either a generic factory or an object-specific factory, and issue an appropriate request on the factory. Then, a new object is created and an object reference is returned to the client. A factory assembles the resources necessary for the existence of an object it creates. Therefore, the factory represents a scope of resource allocation. A finding mechanism, such as a naming context, is used by clients to find a factory. Clients can pass a factory object as a parameter to an operation the client supports.

Class objects in the Legion system are object managers and policy makers. They offer life-cycle service to objects that belong to them. Class objects are responsible for object creating, deleting, copying, moving, object locating and binding. Class objects and CORBA factory objects and factory finders have similar functionality. The Legion_generate_IDL is responsible for mapping life-cycle service between Legion and CORBA. It generates method invocations to those Legion's runtime library functions that related to manipulating objects. For example, CORBA's object creating method in the Generic Factory looks like this:

```
interface GenericFactory {
  boolean support (in Key k);
  Object create_object (in Key k,
            in Criteria the_criteria)
   raise (NoFactory, InvalidCriteria,
            CannotMeetCriteria);
};
```

The key is a name used to identify the desired object to be created. It is defined by the Naming Service. The criteria parameter is expressed as an IDL sequence of name-value pairs that will be extensible. Object can be written that do not interpret the name-value pairs, but just pass them on to other objects.

The Legion object creating code that the Legion_generate_IDL generates looks like this:

```
LOID = Legion.CreateObject(
            class_loid, class_params,
            instance_params);
```

Where the class_loid is the Legion Object Identifier (LOID) of the class object of which the client wants to create an instance. The LOID is an internal object identification mechanism. The class_loid is obtained through a Legion library function that translates a Legion context path name into a LOID:

```
class_loid = Legion.ContextPathLookup(
            class_context_path);
```

The instance parameter lists can be used to transfer additional configuration information into the class object.

4.2 Naming service

The CORBA Naming Service provides the ability to bind a name to an object relative to a naming context. A naming context in CORBA is an object that contains a set of name binding in which each name is unique. Resolving a name determines the object associated with the name in a given context. Binding a context to another context creates a naming graph -- a directed graph with nodes and labeled edges, where the nodes are contexts. Given a context in a naming graph, following the sequence of names will reference an object. This sequence of names is called a compound name and it defines a path in a naming graph. Names are structures, not just character strings. The structure includes a human-chosen string plus a kind field. Graphs of naming contexts can be supported in a distributed, federated fashion. The scalable design allows the distributed, heterogeneous implementation and administration of names and name contexts. Via a names library, name manipulation is simplified and names can be made representation-independent. This allows their representation to evolve without requiring client changes. CORBA's naming services implementations can be application-specific or based on a variety of naming systems currently available on system platforms.

Legion has a three layer naming mechanism for identifying objects. These are: Legion contexts, Legion Object Identifiers (LOIDs) and Legion Object Addresses (LOA). Each object in Legion has a unique LOID but may have more than one LOA. The Legion BindingAgent is responsible for binding a LOID to an LOA. Legion contexts are similar to Unix directories. They can point to other contexts and files as well as any other object in the Legion system, such as hosts, class objects, vaults and tty objects. Command-line operations on Legion contexts look like this:

```
$ legion_ls -l
.               (context)
class           (context)
hosts           (context)
vaults          (context)
home            (context)

$ legion_context_create /home/tmp
Creating context "/home/tmp" in
parent ".".
New context LOID = "1.01.05.608.00..."

$ legion_ls -R
.                  (context)
class              (context)
hosts              (context)
vaults             (context)
home               (context)

In context "home"
   .               (context)
   ..              (context)
   tmp             (context)
```

$

In these examples, legion_ls is similar to "ls" command in Unix, legion_context_create is similar to "mkdir", and legion_set_context is similar to "cd".

LOIDs, which are location-independent strings of bits, are used to uniquely identify Legion objects. Basic LOIDs have fields such as type, domain, class id, instance id, and public key (RSA). The LOA is a list of object-address elements and semantic information that describes how to utilize the list.

The basic naming service features of Legion and CORBA are very similar. In fact, Legion's naming service can be used as a kind of implementation of CORBA Naming Service by the Legion_generate_ID.

4.3 The Legion_generate_IDL

The core part of the CORBA-supporting framework for Legion is the IDL compiler for Legion/C++, Legion_generate_IDL. All C++/IDL mappings and CORBA/Legion object service mapping are carried out by this compiler. The Legion_generate_IDL Legion's IDL compiler is composed of front-end, abstract syntax-tree management and two different back-end parts. The front-end part of the compiler is responsible for lexical and syntactical analysis. The result of front-end analysis is stored into an Abstract Syntax Tree (AST). The back-end generates stub code according to information in AST. Currently, there are two kinds of back-end parts available in the Legion system: the Legion_C++_be and the Legion_java_be. The Legion_java_be only generates client stub files, since the Java implementation of object is not yet supported in Legion.

The Legion_generate_IDL generates Legion-aware stub code. It is responsible for language mapping between IDL and C++ and between IDL and Java. **Figure 8** shows an example of IDL structure mapping. The target class includes not only those mapped data type and auxiliary methods required by CORBA specification but also Legion special data pack and unpack methods.

5. Conclusion

Legion, a large-scale heterogeneous distributed computing platform, is not only capable of handling large-scale computing intensive applications but also has great potential as a platform for large-scale business processing. By supporting the CORBA standard, Legion offers CORBA-based business processing systems and applications a well-scaled heterogeneous distributed platform with a unique capability for harnessing huge amounts of computation and storage resources scattered worldwide.

Future works include implementing more CORBA service mapping in the Legion_generate_IDL, building a CORBA IIOP proxy object in Legion so that Legion can communication with other CORBA OPRs in an effective way, and cooperating with business-processing users to develop practical applications on a CORBA-compliant Legion system.

With the work discussed above, Legion is going to move from a system for traditional computation-intensive applications to a system for data-intensive business processing applications.

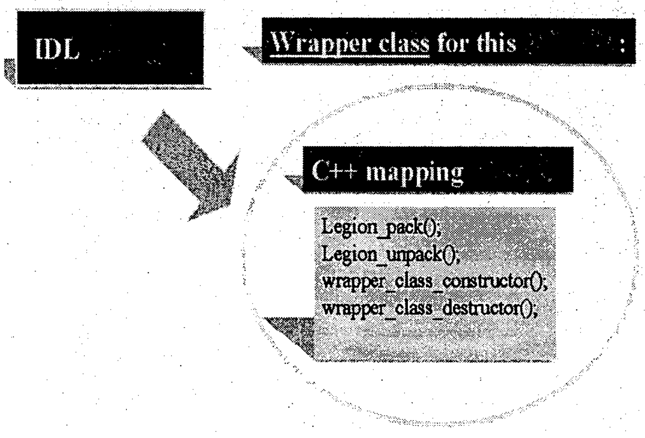

Figure 8. Structure mapping between OMG IDL and Legion/C++

References:

[1] G. Alonso, D. Agrawal, A. El Abbadi, and C. Mohan, *Functionalities and Limitations of Current Workflow Management Systems*, 1997. http://www.almaden.ibm.com/cs/exotica/exotica_papers.html #present.

[2] Andrzej Cichocki, Abdelsalam (Sumi) Helal, Marek Rusinkiewicz, and Darrell Woelk, *Workflow and process automation: Concepts and Technology*, Kluwer Academic Publishers, 1998.

[3] I. Foster and C. Kesselman, "Globus: A Metacomputing Infrastructure Toolkit," *Intl J. Supercomputer Applications*, 11(2): 115-128, 1997.

[4] D. Georgakopoulos, M. Hornick and A. Sheth, "An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure," *Journal on Distributed and Parallel Database Systems*, 3(2) April, 1995.

[5] Paul Grefen, Barbara Pernici, and Gabriel Sanchez, *Database Support For Workflow Management*, The WIDE project, Kluwer Academic Publishers, 1999.

[6] A. Grimshaw, A. Ferrari, F. Knabe, and M. Humphrey, "Legion: An Operating System for Wide-Area Computing," Technical Report CS-99-12, University of Virginia, March, 1999.

[7] Andrew S. Grimshaw and Wm. A. Wulf, "Legion--A View From 50,000 Feet," *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, IEEE Computer Society Press, Los Alamitos, CA, August 1996.

[8] M. Kamath, G. Alonso, R. Guenthoer, and C. Mohan, "Providing High Availability in Very Large Workflow Management Systems," *Proceedingsof the Fifth International Conference on Extending Database Technology*, Avignon, March 1996.

[9] Bernd Kramer, Michael Papazoglou and Heinz-W. Schmidt, *Information Systems Interoperability*, Research Studies Press LTD, 1998

[10] Legion Group, *Legion 1.6 Developer Manual,* Oct. 1999. http://legion.virginia.edu/documentation.html

[11] Michael J. Lewis and Andrew Grimshaw, "The Core Legion Object Model," *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, IEEE Computer Society Press, Los Alamitos, CA, August 1996.

[12] Greg Lindahl, Andrew Grimshaw, Adam Ferrari, and Katherine Holcomb, "Metacomputing--What's in it for me?," http://legion.virginia.edu/papers.html.

[13] J. Miller, A. Sheth, K. Kochut and X. Wang, "CORBA-Based Run-Time Architectures for Workflow Management Systems," *Journal of Database Management, Special Issue on Multidatabases*, Vol. 7, No. 1, pp. 16-27, Winter, 1996

[14] C. Mohan, "Recent Trends in Workflow Management Products," *Standards and Research, Proceedings of the NATO Advanced Study Institute (ASI) on Workflow Management Systems and Interoperability*, Springer Verlag, 1998.

[15] Thomas J. Mowbray and Willm A. Ruh, *Inside CORBA: Distributed Object Standards And Applications*, Addison-Wesley, 1997.

[16] C. Mohan, G. Alonso, R. Guenthoer, M. Kamath, and B. Reinwald, "An Overview of the Exotica Research Project on Workflow Management Systems," *Proceedings of the Sixth International Workshop on High Performance Transaction Systems*, Asilomar, September 1995.

[17] Object Management Group, *CORBAservices: Common Object Services Specification*, Nov. 1997.

[18] Marc-Thomas Schmidt, "The Evolution of Workflow Standards," *IEEE Concurrency*, pp. 44-52, July-Sept. 1999.

[19] Jon Siegel, *CORBA Fundamentals and Programming*, John Wiley & Sons, Inc, 1996.

[20] Charles L. Viles, Michael J. Lewis, Adam J. Ferrari, Anh Nguyen-Tuong, and Andrew S. Grimshaw, "Enabling Flexibility in the Legion Run-Time Library," *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'97)*, pp. 265-274, Las Vegas, Nevada June 30-July 2, 1997.

[21] S.M. Wheater, S.K. Shrivastava, and F. Ranno, "A CORBA Compliant Transactional Workflow System for Internet Applications," *MIDDLEWARE' 98*, The Lake District, UK, September 15-18, 1998.