A Geographically Distributed Processing Environment for Intelligent Systems

Charles Hannon
Computer Science Department
Texas Christian University
Fort Worth, Texas, USA
C.Hannon@tcu.edu

ABSTRACT

By addressing both the theoretical and practical issues of a geographically distributed application domain, we propose a way to construct a processing system using the computational aspects of a brain/mind system. The Alchemy distributed processing environment has been developed to support the geographical distribution of a set of intelligent applications running over the Internet. We focus here on the implementation details of the Alchemy architecture and how its performance has been demonstrated using a model of a geographically distributed information system. Using this model we compare our approach to more traditional approaches like distributed objects and mobile agents.

Keywords: Special Purpose Architectures, Security and Authentication

1 INTRODUCTION

Geographically distributed processing environments have evolved from parallel processing emulation (e.g., PVM, MPI, etc.) and traditional networked applications (e.g., distributed databases and file-sharing, network-enabled HMIs, etc.). These environments are based on either a tightly-coupled cluster of processing elements (e.g., Beowulf) dedicated to a set of carefully controlled tasks, or a grid of very loosely connected processing elements (e.g., Legion or Globus) that allow program-level tasks to compete for available resources. While there are advantages to using either approach as the distributed processing layer of an intelligent system, their disadvantages merit the examination of computational models that fit somewhat in the middle of these two extremes. Using what is known of the workings of the brain/mind computational system, our Alchemy system attempts to build a distributed processing domain that supports a more loosely connected environment than a traditional cluster while providing more resource-wide process control than most current grid approaches.

Alchemy builds on what was learned using a more specialized Adaptive Multi-agent environment for Explanatory-Based Agents (AMEBA) system [5]. Unlike pure connectionist approaches, Alchemy does not try to simulate a neural-network, but draws a processing analogy from the brain's design and construction. This not only reduces the resulting complexity of the environment but allows us to incorporate other important features like secure communication, GUI-based application control and load balancing.

In the next section, we will further justify using our own architecture for the distributed basis of our research by providing an overall contrast between it and related systems. We will then discuss how Alchemy's concurrency model, security mechanisms, and other aspects relate to an exemplary distributed object and mobile agent approach, these being OMG's CORBA and IBM Aglets. Finally, we define an experiment for testing Alchemy's performance against the CORBA and Aglets environments and discuss the results of this experiment.

2 RELATED SYSTEMS

Alchemy has been designed for intelligent applications that have critical time relationships between components. While some Alchemy applications can be run on a cluster, others (like remote smart home monitoring and control) need to be geographically distributed. These applications are also poorly suited for grid architectures which best handle the batch-like processing done on a supercomputer. Alchemy is by no means the only distributed environment attempting to fill the middle ground between clusters and grids. Even early research on distributed operating systems like Amoeba [9] and distributed shared memory systems like Linda [2] had the goal of a tightly-controlled distributed domain running on a loosely-coupled processing environment, but the low-level nature of these systems failed to deliver on such a promise. More recently, distributed object environments like CORBA [7] and DCOM [3] have attempted to seamlessly extend object models across processors, but these approaches suffer from the overhead of a generic approach to the object distribution problem. Even with new process paradigms, like the mobile agents systems of IBM Aglets [6] and Sun's JINI [4], their success is affected by the complexity of a distributed approach using a traditional computation model.

To function in a geographically distributed domain, both the distributed operating system and distributed

shared memory approaches need to unify portions of the Internet into virtual computational systems. While completely hiding the underlying communication and security difficulties from the application level, the resulting approach must maintain a high degree of data concurrency between all of the various processes or threads of all the applications running under that environment. Even with careful assignment, monitoring and migration of all of the threads of execution of all of the supported applications, these approaches would be highly susceptible to load mismatch in a loosely connected environment where the communication latency is for the most part non-deterministic.

One approach to reduce the need for system-wide understanding of processing state and data concurrency is to break an application into a collection of objects or agents and allow either the application as a whole or its components to distribute themselves. Such applications can then decide for themselves which data elements need to be globally shared and can somewhat control their own process concurrency by controlling where objects or agents are located in the processing domain. In a geographically distributed object approach, it is theoretically possible to reduce the concurrency problem down to the way methods are called, since the execution of a task can be fully contained in a known sequence of method invocations and the resulting execution and communication time need to temporally traverse this sequence. However, there are normally far too many possible sequences at any given state of an application for them to be used as a predictive mechanism of the total processing load of the distributed application even if the communication latency between computation nodes could be characterized. This can be somewhat overcome by using something like a name service to clone additional objects of a given type in additional threads of execution when it is determined that a method in the sequence has formed a processing bottleneck.

Mobile agents can theoretically further improve the concurrent performance of an application by allowing pieces of the application to distribute, clone and/or redistribute themselves based on overall processing load. However, both the object and agent approaches have little or no way of balancing the demands of the overall processing environment since they result from a divide-and-conquer approach which is inherently greedy. For example, in creating a new set of server-side objects in CORBA to improve task execution or moving a set of Aglets to improve the system's overall communication load, either to new objects or mobile agents can cause one or more computational nodes to be become so unbalanced that there is no overall system gain.

To overcome these limitations with the existing approaches discussed above, it is necessary of reevaluate our whole approach of dealing with a geographically distributed domain. The massive number of nodes making up the Internet's processing resources and the loose communication structure making up its connection mechanism make it quite different from a traditional set of tightly coupled processors found in parallel message-passing, SMP, or cc-NUMA architectures. When looking at the time-critical aspects of geographically distributed intelligent systems, any further extension of the sequential processing model may simply not be applicable, and thus, we propose revisiting a wetware-based analogy. When you consider enough of its computational nodes, the Internet forms a processing graph which is very similar to a logical representation of parts of the brain. Further, the range and rate of communication speeds in the brain are very similar to that of the Internet [1].

3 CONCURRENCY

Unless an application is being distributed to simply improve access to distributed data sources (e.g., the World Wide Web, FTP, etc.), there is a clear expectation that the distribution of an application across multiple processing nodes will result in some level of speedup. However, a number of factors may result in a distributed environment being inherently poorly suited for its applications to gain the maximum benefit from concurrent operations. CORBA, for example, was originally designed to provide a simple object-based connection mechanism for legacy software, and therefore, was designed around a Remote Procedure Call (RPC) communication mechanism. RPC is inherently sequential in operation, with a client making a call to a server via the remote procedure, and then, waiting for the return value of this call. While the designers of CORBA have always understood this as a limitation and from the very beginning of its standardization took steps to incorporate concurrent mechanisms in CORBA, the environment is still not very concurrency friendly with the application designer being forced to supply most of the multi-threaded aspects via raw thread calls.

To thread a CORBA application both the server and client side of every RPC connection must be addressed. On the server side, most implementations of CORBA allow the designer to automatically spawn a new server thread to support the client's execution of the RPC via a name service, but this mechanism works best (and sometime only) in a SMP type of environment since ORBs do not normally extend across processor group boundaries. The application designer is, of course, free to thread a server herself, but in doing so, the way the IDL is defined and used by the ORB can often make this difficult. As a general rule, most ORBs (due to the way the processed IDL definition relate to the data storage used for parameter and state maintenance) only support a channel-based threading model and do not allow the

threading of each transaction. On the client-side, the problem of threading is even more difficult since all RPCs in CORBA stall the calling thread and threading each and every client-side RPC in an application can be problematic at best since the internal data within a RPC is not always stored in a thread-safe manner.

IBM Aglets support a very rich set of message types, including both synchronous and asynchronous messages, as well as, some rather strange synchronous message types that do not block the caller but schedule return messages to be sent in the future. While fixing the obvious problems with the RPC type of messaging, Aglets introduce a new concurrency difficulty in that they are built on top of Java. This means that an Aglets application is not directly sending messages to the network interface, but to the Java Virtual Machine (JVM). Newer versions of Java (in fact, since version 1.1.8) support both green (Java's original user-level) threads and native threads (which are system-level threads on most system). Even if the JVM is set up to use native threads, Java's thread interface runs very slowly. The test results presented later in this paper will show that this performance problem greatly affects an Aglets application's concurrent operations.

Unlike either CORBA or Aglets, Alchemy attempts to hide as much of the threading details as possible from the application designer. Being based on a brain model, Alchemy does not natively support the concept of traditional sequential processing, and therefore, does not support any type of synchronous messaging. Instead, the Alchemy architecture divides an application into a set of asynchronous processing elements (called message handlers) that process messages received from other processing elements via either a client-side or server-side connection. The only difference between these connection types is that each server-side connection can be connected to multiple client-side connections. Each connection instance in an Alchemy application runs in its own process thread within a heavy-weight container called a node. All nodes, connections and handlers are accessed via a name service. This name service hides all of the connection and threading details of an application from the designer. To build an application, the designer simply builds a graph of nodes using a GUI interface to define its nodes and connection arcs, and then, defines all of the handlers needed to do the work of the application. Once this is done, the Alchemy support layer is responsible for distributing the application and balancing its load with all of the other applications running on the defined set of processors.

4 SECURITY

Most grid systems are built on top of something like the openSSL security libraries [8]. Neither CORBA nor any other distributed objects domain was originally designed to support any security mechanism. While there is a great deal of interest in providing both communication and process control security to CORBA, these efforts are severely limited by some early design decisions. Unless a secure CORBA is being used, CORBA provides little or no support for adding security mechanisms at the application layer. For example, while message encryption can be added to an application by simply encrypting the data before it is sent through the RPC, this process basically breaks the way IDL is designed to handle the passing of data of various types between distributed objects and each server-side RPC implementation would have to explicitly handle the decryption of input and encryption of output. Secure CORBA uses SSL as its transport layer and makes other security changes to the ORB, but this use of SSL greatly impacts the overall performance of a CORBA application in the process.

IBM Aglets, on the other hand, was designed around the built-in security mechanisms of Java to provide a complex set of security parameters for controlling what a mobile agent can or can not do once it arrives at a processing node. This capability was introduced into Aglets based on a rather optimistic view of how agents would travel around an unconfederated network doing their work on computers not managed or even controlled by the party or parties owning the Aglets application. Since Aglets were primarily designed to support the fetching of data by the agent by moving to the source and then back to its point of origin, very little thought has been put into securing the way data is passed between agents. The results of these design decisions is a complex security mechanism which is actually not that useful.

Alchemy uses a common communication interface for both system-level and application messages and supplies a three-level security method which includes communication channel authentication, encryption and message tracking. This communication interface is socket-channel based (like SSL), but unlike SSL has been highly tailored to improve connection time in a dynamic environment with many related clients and servers. The Alchemy security mechanisms are primarily designed to: (1) provide a very secure communication method for process control messaging, (2) allow the application designer to decide how much communication security they want to provide for application messages, and (3) provide a test bed for further network security enhancements.

The lowest security level (level 1) uses a four-way authentication method to confirm that both the client and server are in fact Alchemy components. A four-way authentication method is used because the Alchemy key management system shares these messages for passing the

information used to construct the first symmetric session key used by a message channel. Thus, the authentication protocol is also encrypted using a 'pre-session' key (a symmetric key created and maintained like a session key but only used during authentication to remove the need for a second public key). At level 2, the encrypted of all message content is added using randomly rotating symmetric keys. At level 3, a message tracking mechanism (currently built on MD5) is added. All system-level support traffic (e.g., GUI to process control servers, server-to-server, etc.) is locked to level 3. The security level of all other traffic can be selected by defining the security level of each server during the application design. The location of the data in a level 1 and level 2/3 message is shifted so that one cannot easily gain information about message construction by sending the same message both encrypted and in the clear.

Alchemy uses both RSA public and Blowfish symmetric encryption to form a three-step key creation process which is both very secure and relatively fast. To avoid the need to lock box a super-secret key, each server creates a 1024-bit RSA public key at startup. To improve both security and speed, this public key is only used once per channel session and is never used to authenticate the server. When a client attempts to connect, it is sent the modulus part of the public key. It then encrypts a 256-bit 'pre-session' key for use during authentication. Encoded in the authentication stream are pieces of the first session key, so if both the client and the server pass authentication, they both end up with the first session key. This key is then randomly changed during the session to further improve security.

5 OTHER ASPECTS OF ALCHEMY

The Alchemy architecture in many ways functions like a purely object-oriented environment, like Smalltalk, where class methods are called by event messages, not by direct invocation. However, Alchemy is inherently distributed and totally dynamic. Concurrency is not an optional feature but the only mechanism by which an application can be implemented. The number of distributed nodes in an application, the number of server and client connections maintained by a node, and the mapping between handlers and connections can all be changed at any time during the application's life. Thus, an application can dynamically reconfigure any application's processing environment or completely change its or another application's functionality while they are running.

As depicted in figure 1, Alchemy uses the concept of nodes to support the distribution of processing elements across a set of computational resources. The encapsulated message handlers of these nodes are abstracted from their and other handlers' physical locations by a name service which (1) maps client connections, server connections and handlers to nodes and (2) maps nodes to physical processors. Process mobility is accomplished by simply moving a connection and its associated handler to a new node in the system. This allows Alchemy to more tightly manage its processing elements than most mobile agent and distributed object approaches while providing a better capacity to handle the dynamic nature of the Internet. Due to this flexibility, an application's processing elements are more loosely coupled than a traditional distributed system. While adding some complexity in application design, this statistically overcomes some of the jitter in the communication latency experienced by a geographically distributed domain.

Applications under Alchemy use a processing model that allows individual handlers to run completely independent from other handlers. A handler listens for messages from other handlers, and then, performs asynchronous actions that most often create new messages. Using a broadcast or multicast communication method, each handler becomes a completely isolated process that accepts only the messages it understands and has time to handle. This results in a far more non-deterministic process flow where trails of execution can fork, merge and die without causing such traditional problems as deadlocks and race conditions. Further, any number of handlers can be ganged together as a service group, thus allowing any jitter on the communication latency to be overcome by the geographic distribution of the required service.

The Alchemy environment has no separate support layer. Process control is handled by two special types of nodes, the Alchemy Server and the Generic Alchemy Definition, Generation and Evaluation Tool (GADGET). These tools allow nodes to be started, stopped and moved to any processor (or cluster) during application execution. They also allow the individual control of the client and server connections and the assignment of handlers to connections. The GADGET also supports a GUI

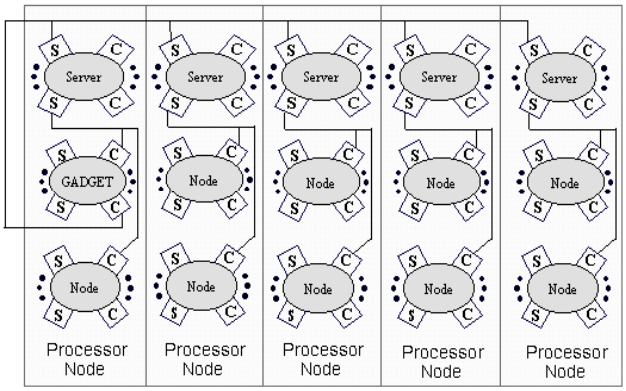

Figure 1. Alchemy's Process Architecture

construction tool for designing applications and a trace mechanism for application testing.

Another special node type, the Alchemy Superserver, is used to map the function of servers to a named service type. The Superserver is most similar to the way JINI supports service connections, but also has similarities to CORBA's name service. An Alchemy client can chose to connect to a particular server or simply a particular service type. For example, if client (App1: Node1:Client1) needs to communicate with (App1: Node2:Server1) or (App1:Node3:Server2) which both provide service (App1:Service1), then the application designer can select the connection using either the logical location (e.g., App1:Node2:Server1) or the service type (i.e., App1: Service1).

An important aspect of any distributed object or agent environment is the mechanisms used for component communicate. For example, CORBA uses its IDL to define a set of data types, and then, handles the marshaling of these types across the IPC communication channels created by the ORB. Most implementations of CORBA select the IPC method of the communication channel based on the relative location of the client and server. CORBA also provides a name space and name service to allow some level of location abstraction.

Alchemy uses a very basic form of data marshaling which converts all data passed between nodes to their string representation. This method was selected due to its cross-platform simplicity and compatibility with block encryption methods. The tested version of Alchemy only uses Internet (or Berkeley) sockets, but other versions select the IPC method based on node locality. Alchemy's name space crosses application boundaries. This allows multiple applications to run on the same set of Alchemy servers and provides better load balancing across multiple distributed applications running on common resources. The mapping between the logical location and physical location (i.e., the host processor or cluster) of a node can either be stored as part of the application definition or dynamically selected using the GADGET.

6 THE TESTING ENVIRONMENT

A number of parallel/distributed benchmarks are available for anything from low-level system calls to high-level functional tasks. After examining a number of these benchmarks, it was determined that none of them were particularly well suited to evaluate Alchemy's unique ability to handle geographically distributed intelligent applications. Therefore, we developed our own simplified testing application which was constructed to formally capture the search domain of a generic Internet-based information system. This new 'benchmark' was designed to reflect the processing load of an intelligent system without containing any of the programmatic complexity of a truly intelligent system.

The domain consisted of: (1) data element types labeled *A* through *Z,* (2) result types labeled *aa* through *dz,* and (3) a set of mapping rules of the form:

$$rr \leftarrow D_1, [D_2, \ldots D_n] \text{ (rel n.nn)}$$

where the left-hand-side (*lhs*) consists of a single result type and right-hand-side (*rhs*) consists of a conjunction of any number of data element types and a simple relevance factor for the rule.

The test system consists of five component types, each of which can be replicated as many times as needed to support the domain instance. These components are: (1) a *Data Consumer* that can execute any query (i.e., a request for a result) and relevance order the results, (2) a *Data Mapping Index* that knows the location of all mapping rules for a single query type, (3) a *Data Mapping Device* that contains a single mapping rule that it can execute upon request, (4) a *Data Storage Index* that knows the location of all instances of one of the data element types, and (5) a *Data Storage Device* that stores any number of instances of any number of data element types. To simplify testing, each instance of a data element is labeled D_{mn} where *m* is a number that uniquely identifies the Data Storage Device on which the element resides and *n* is the number of the instance on that device that is being addressed.

Any Data Consumer in the system can initiate a search by sending a query to a Data Mapping Index. The Data Mapping Index then locates all matching Data Mapping Devices and sends a request for them to resolve all rule matches. A Data Mapping Device then requests the location of any Data Storage Devices containing relevant data from the Data Storage Index and requests the relevant data from these devices. The Data Storage Devices then send a list of all data elements matched to the requesting Data Mapping Device which creates a complete list of all rule matches. The rule matches are sent to the requesting Data Consumer which combines the lists and orders the results based on relevance.

7 RESULTS

The test application has been implemented using three different distributed environments: (1) CORBA (Orbit and others), (2) IBM Aglets (running under Java) and (3) Alchemy. Testing was performed on a Beowulf-like 16 processor (2x8) SMP cluster running Linux. To simulate something closer to the Internet, the network channel bonding of this system was turned off and only one of its three 100baseT networks was used. This network was also connected via a hub, instead of its normal switch, reducing the maximum network throughput to 100Mbits/sec.

Figure 2 depicts the average execution time of a set of 25 runs on two different data sets. As can be seen from this graph (which uses a logarithmic scale), the Alchemy-based application demonstrates better performance than the CORBA-based application and much better performance than the Java-based application even when the JVM is using native threads. Figure 3 presents the speedup calculations from the raw data given in figure 2. Again, Alchemy demonstrates the best speedup performance and Java again demonstrates the worst level of speedup. Both the Alchemy and CORBA get message bound when the application is distributed across 16 processors, but this is a little misleading for the Alchemy implementation since the overall design of the application had to be somewhat serialized to accommodate CORBA's RPC message mechanism and we had problems getting the CORBA application to handle larger databases where this message binding would not be as much of a factor.

8 CONCLUSION

The test application shows that Alchemy is faster in overall performance and demonstrates better speedup than either IBM Aglets or CORBA for very simple intelligent applications while improving the communication security and local control of the application. Past work with AMEBA has shown that extending the intelligence of the application improves its speedup. Therefore, our decision to create our own special purpose distributed domain has been somewhat justified by the results presented here.

One concern introduced by our results is the fall off in speedup seen when the test application is distributed across a large number of processors. Any application will at some point reach processing saturation and this will always occur faster with coarse-grain parallelism like that being used in Alchemy. While further testing of Alchemy is needed to completely understand its potential as a more generic distributed processing environment, the results so

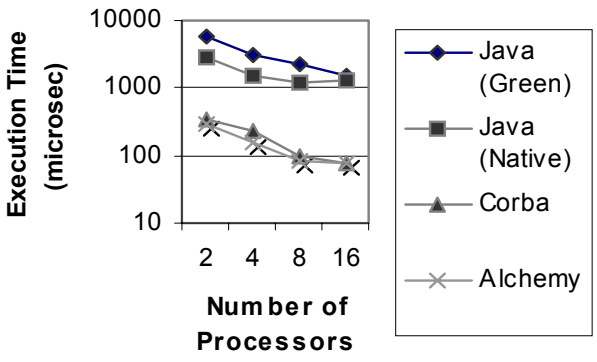

Figure 2. Execution Time in Microseconds

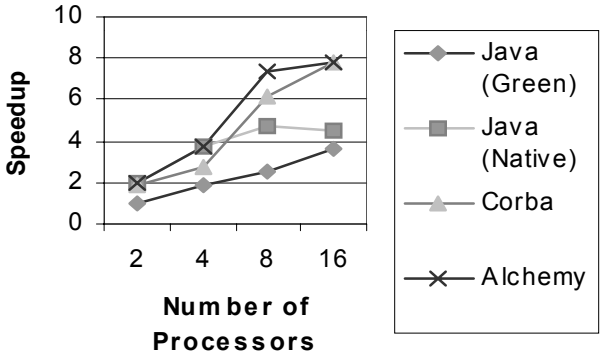

Figure 3. Speedup Calculations

far indicate that it might be able to overcome some of the difficult problems encountered in a geographically distributed domain while still providing some level of speedup. Given that it can support a number of simultaneously running dynamic applications, this type of processing model could potentially improve how the limited resource of wide-area network bandwidth are used by a number of competing applications.

9 REFERENCES

[1] Anderson, J. R. *Cognitive Psychology and its Implications*. New York: W. H. Freeman and Company, 1995.

[2] Bjornson, R. *Linda on Distributed Memory Multiprocessors*. Ph.D. Dissertation, Technical Report 931, Yale University, Department of Computer Science, Nov. 1992.

[3] Eddon G. and H. Eddon. *Inside Distributed Com.* Microsoft Press, 1998.

[4] Edwards, Keith W. *Core Jini*. Sun Microsystems Press. 2000.

[5] Hannon, C. and D. J. Cook. "Developing a Tool for Unified Cognitive Modeling using a Model of Learning and Understanding in Young Children." *The International Journal of AI Tools,* 10 (2001): 39-63.

[6] Lange, D. B. and M. Oshima. *Programming & Deploying Mobile Agents with Java Aglets*. Addison-Wesley Pub Co., 1998.

[7] Mowbray, T.J. and R. Zahavi. *Essential CORBA, The: System Integration Using Distributed Objects*. John Wiley & Sons, 1995.

[8] *OpenSSL: The Open Source toolkit for SSL/TLS*, http://www.openSSL.org, 2002.

[9] Tanenbaum, et al. "Experiences with the Amoeba Distributed Operating System." *Communications of the ACM*, 33 (1990): 46-63.