

Architecture of a Distributed Imaging System

Ian R. Greenshields and Z. Yang
Department of Computer Science and Engineering
University of Connecticut

Abstract

We describe the architecture of a distributed medical imaging system. Designed to support both code and data mobility, the system attempts to exploit available computational resources within a collaborating group to apportion imaging tasks to the platforms best suited to their completion. The system is designed around a mixed Java-RMI, CORBA and Agent architecture. The present paper surveys various features of the system.

1: Introduction

The standard mechanism for delivering value-added imaging services to a clinical environment has been the design and emplacement of single-platform imaging systems within the environment. The difficulties and problems associated with this approach are legion:- most obviously, the software will run only on the platform(s) it has been written for; the authors must express not only the imaging algorithms, but also relate these to the platforms OS for windowing and file management; changes to the software are tedious, and in an academic environment, often result in wholesale rewriting of the package; and porting the software to a new platform (even when the base OS on the new platform is reputedly a variation of the OS on the old platform – one thinks of Unix varieties) is at least tedious at best, and if the software makes extensive use of a windowing system, often a problem of the same magnitude as the rewriting of the package in its entirety for the new platform. Finally, upgrades to the software (either corrective or additive) require notifying each and every client using the system to ensure that the platform is homogeneous over its clientele. An attractive (and obvious) solution to the platform dependencies (particularly as they apply to window management etc.) is to exploit the ubiquity of the web browser, deferring all window (and indeed local file as well as more general GUI) management to the client's browser. This approach can be adopted in a number of ways. One is to centralize imaging algorithms at a single server, producing a conventional client-server architecture, with client-server interactions implemented by (say) HTML/CGI [1]. This is the architecture of the OIS imaging system at the University of Connecticut [2]. While such client-server systems allow for ease of algorithm maintenance, they are resource-limited (the server rapidly becomes a bottleneck) and, if expressed in HTML/CGI, inelegant and slow [3]. The simple delivery of Java to the client is also not a complete solution; even although JIT compilation significantly improves Java's performance, there will remain many important, large imaging algorithms (such as MRF/GRF classifiers) which are not amenable single-platform execution (Java or otherwise).

Our goal is to provide a comprehensive, evolving imaging facility which can be incrementally improved by any participant in the facility, is platform independent, attempts to distribute loads as it deems necessary and is centrally manageable. To accomplish this, we have

elected to design the architecture around the notions of local/remote cluster computing supporting an agent, Java-RMI and CORBA middleware architecture.

2: Architecture: Overview

2.1: Physical Architecture.

We consider a collection of distributed machines, typically workstation-level boxes running an appropriate (supporting) OS. We distinguish between three classes of machines. *Strong clusters* are groups of machines organized into a cluster computer supporting a conventional parallel working environment (such as MPI or PVM [4]). Typically, these machines will be in close proximity to each other and connected through some high-speed fabric (such as ATM). Machines may also belong to a *weak cluster* i.e., machines (not necessarily in proximity to each other) which support a weaker form of parallel interaction, such as machines which are agent-enabled. Finally, machines may simply belong to the general distributed architecture, capable of supporting services they might offer via RMI or CORBA, and capable of hosting mobile code such as Java, but not willing to be exploited for general parallel/distributed computation.

2.2: Middleware Architecture.

The basic protocol/middleware architecture is the familiar CORBA/Java-RMI architecture [5] [6] [7] [8]. CORBA and the CORBA-Java connection are now so commonplace that no space here need be taken in describing their deployment within this context. We note the following, however: our preference is to deploy RMI for simple Java-to-Java call-backs (clients to servers) or, if the platform supports it, a Java-specific CORBA ORB rather than a generic ORB (e.g., Netscape's Caffeine [9]). The middleware architecture of the system in as much as it relates to the Java-CORBA model is essentially standard.

2.3: Client Architecture.

A primary goal of the system is to permit easy extension of imaging services, to be provided by any of the architecture's participants. Within the Java/CORBA framework, the most obvious model for accomplishing this is the Dynamic Invocation Interface and the general metastructure which surrounds the DII. The primary responsibilities of the client-side architecture are to maintain and track the client's image condition and state, and to negotiate for services provided by other clients in their server roles. The negotiation function—the primary function—relates to the trader architecture as described in the Open Distributed Processing specification in [10]. Traders are objects which fulfill three essential functions: they serve as a repository of services offered by clients within the distributed environment; they accept and respond to requests for services made by clients, and they identify to the requesting client the location of and specifications for the service requested by the client should such a service exist. The simplest implementation of the trading function has one identified trader which serves the entire distributed system, but since this violates simple scalability and reliability issues, we have opted to implement traders in their *federated* mode relative to the strong/weak cluster architecture described above. Thus, machines configured in a weak/strong cluster will share a trader. In turn, these traders are linked in a federation so that the offer space of each individual trader is enlarged by its federation with other traders. Aside from implementation issues, the most obvious

ramification to this is that both weak and strong clusters be relatively small in size (so that local traders are not overwhelmed by the number of their clientele); as it happens, this scalability issue derived from the trading function merges well with general observations on the suitability of small clusters for parallel programming [11].

It is important to recognize that services advertised remotely may not in fact be executed remotely. Since many simple imaging tasks (such as convolution filters) represent compact code and can execute efficiently on a modern PC, it seems absurd to transfer significant volumes of image data over the network to a remote class. Thus, many “remote” classes will in fact be nothing more than vehicles for the delivery of applets back to the requesting (local) client.

If the local client does not export services, then the major function of the client is essentially to seek out and list services being advertised within the distributed imaging environment. However, two additional functions may be expected of local clients. In a fully collaborative environment, we expect that services developed locally will be written to conform to the Common Image Description Protocol (see below), so that they are exportable and usable by other clients in the distributed environment. However, even if local clients are not capable of exporting their own services, they may nonetheless be expected to serve as weak or strong hosts for parallel activity running over the network. We address this next.

3: Weak and Strong Hosts

3.1: Weak Hosts.

A client is said to be a *weak host* provided it is willing to share a parallel workload within a restricted, secure environment. (Initiating servers have *weak* access to the client). Java applets execute in a weak environment, and the weak structure we implement is that of the java agent known as the *aglet* [12] [13]. Provided a host is aglet-enabled, workload from a machine in its weak-cluster hierarchy can be offloaded to it. The two situations under which this can happen are: a client within the local weak cluster (hierarchy) downloaded an applet which is a parallelizing agent (i.e., a piece of code which seeks to distribute itself over the members of the weak cluster), or a client within the local weak cluster (hierarchy) received a remote request for service which it desires to execute as a parallelized agent within its own weak cluster. However, we specify that aglets not simply be shipped to aglet-enabled hosts for immediate execution. Instead, the weak parallel activity employs at least two classes of aglets (a third is under consideration). These are *diplomat aglets*, whose duty is to negotiate resources and rights for worker aglets, and *worker aglets*, which in fact carry out the computations. Hosts may reject workers explicitly (for example, if the workstation is in heavy personal use), or a host may be by-passed if the aglets determine that the *javaIntsPerSecond* or *javaFloatsPerSecond* self-benchmarking tests made by applets in the system fall below a threshold carried by the applet. (These are functions not only of the underlying JVM, but also of the scheduling on the system).

Access rights within a host in a weak cluster must (as noted above) be prenegotiated by the diplomat aglet associated with a worker task. Aside from provisioning processor resources, a host may be asked if it is willing to concede file-space rights. The most likely request from a remote aglet will be for *write privileges* on the host’s local file system, usually to maintain image, processed image and/or state data between successive computations. The kind of security architectures needed when applets play out of their restricted sandbox has been addressed by the Princeton Secure Internet Programming group [14]. Of particular

interest to us is the notion of name space interposition, which is a technique whereby safer (more restricted) versions of a class are interposed either by the originating server (if it owns the class hierarchy concerned and has been informed by the diplomat that there are resource restrictions) or by the local host (if it owns the appropriate class hierarchy).

3.2: Strong Hosts.

A host is a *strong host* if it is configured to accept conventional parallel computation using a conventional mechanism such as PVM or MPI. Hosts may advertise their willingness to support tight parallel activity via the trader object. The security mechanisms erected both by native operating systems and system administrators effectively rule out dynamic assignment to strong clusters. Rather, strong clusters advertise pre-compiled services which execute in parallel on the cluster as against volunteering their services to arbitrary MPI/PVM code. For example, we offer a Gibbs/MRF classifier which runs in parallel on a (strong) 8-node cluster in (compiled) C++ and MPI.

4: Common Image Description Protocol

The facility we describe is specifically designed to handle image processing tasks. The notorious wide variations in image format and encoding necessitate that remote classes all handle images in exactly the same way. The obvious way to accomplish this is to gate images through a common image description data structure. Classes which act on images do so through the structures held in the Image Control Block (ICB), and must access the ICB through its reader/writer mechanisms. In addition, however, to simply describing the format of a given image, the ICB also maintains *state* information regarding the image. It tracks, for example, the number of accesses to the image; whether the image can be moved; the location of the last derived image; the location of the last classified image; information about the class structure in the image; how the image is to be acted upon (if, for example, it is multispectral or volumetric); what compression strategies are acceptable for the return of remotely computed (derived) image data based on this image etc. For example, a image sharpening applet arriving at a client may be informed (by the ICB associated with the upcalling process) that the image to be sharpened is a volumetric, multispectral image which is to be sharpened only in spectrum S_i of plane P_k ; that if needed, the image can be transported elsewhere, and that any remote results can be returned in lossy compressed form back to the client.

5: Acknowledgements

This work is funded by a grant from the State of Connecticut under its Critical Technologies Project. The authors acknowledge the on-going support of the Department of Radiology of the University of Connecticut's Health Center.

6: Literature Cited

References

- [1] Umar, A. *Object-Oriented Client/Server Internet Environments*, Prentice-Hall, Upper Saddle River, N.J., 1997

- [2] *OIS Imaging System*, Department of Computer Science and Engineering, University of Connecticut, 1997
- [3] Orfali, D. and D. Harkey: *Client/Server Programming with Java and CORBA*, Wiley, New York NY, 1997
- [4] Kumar, V., A. Grama, A. Gupta and G. Karypis: *Introduction to Parallel Computing*, Benjamin Cummings, Redwood City, CA, 1994
- [5] Vinoski, S. CORBA: Integrating Diverse Applications within Distributed Heterogeneous Environments, *IEEE Communications Magazine*, Feb. 1997, pp. 46-55
- [6] Evans, E. and D. Rogers: Using Java Applets and CORBA for Multi-User Distributed Applications, *IEEE Internet Computing*, May-June 1997, pp. 43-55
- [7] Hamilton, M.: Java and the Shift to Net-Centric Computing, *Computer* **29** No. 8, Aug. 1996, pp. 31-39
- [8] Carlson, A., W.R. Brook and C.L.F. Haynes: Experiences with Distributed Objects, *AT&T Technical Journal*, March-April 1996., pp. 58-66
- [9] Netscape Communications Inc., Netscape Internet Service Broker for Java, at <http://developer.netscape.com/docs/manuals/enterprise/javapg>
- [10] ITU/ISO, "Reference Model of Open Distributed Processing - Part 3 :Architecture", ISO/IEC 10746-3, ITU-T Rec. X903, (1995)
- [11] Pfister, G.: *In Search of Clusters*, Prentice-Hall, 1995
- [12] Lange, D.B., M. Oshima, G. Karjoth, and K. Kosaka: Aglets: Programming Mobile Agents in Java, *proceedings of Worldwide Computing and Its Applications (WWCA '97)*, Lecture Notes in Computer Science, Vol. 1274, 1997
- [13] Karjoth G., D. B. Lange, and M. Oshima: A Security Model for Aglets, *IEEE Internet Computing*, May-June 1997, pp. 43-55
- [14] Wallach, D., D. Balfanz, D. Dean and E.W. Felten: Extensible Security Architectures for Java, *16th Symp. on Operating Systems Principles*, Saint-Malo, 1997. Also <http://www.cs.princeton.edu/sip/pub/sosp97.html>