Janet's Abstract Distributed Service Component

Peter Cappello
Department of Computer Science
University of California, Santa Barbara
Santa Barbara, CA, USA 93106
email: cappello@cs.ucsb.edu

ABSTRACT
We motivate the value of an abstract distributed service component (ADSC) with an overview of JANET, a high-performance grid computing service. We then present the architecture of an ADSC, focusing on some design issues.

KEY WORDS
Distributed computing, Grid, Java, Object-oriented design

1 Introduction

Distributed systems are difficult to design and implement correctly. Object-orientation simplifies the process, but does not entirely alleviate this difficulty. Reusable components improve developer productivity. Abstraction leads to reusable components. We contribute an abstract distributed service component (ADSC), that we hope is suitable for reuse by others. We also hope our presentation of some design issues will spare others some of the our missteps, even if they ultimately prefer a different design.

Space limitations prevent a detailed placement of this work in the context of others [7, 5, 17, 10, 16]. Please see [13] for some references to distributed computing work that predates object-orientation; some object-oriented work includes [8, 2, 1, 3, 12, 14, 4]. Ibis [15]gives an insightful view of middle ware interfaces for a Java-centric distributed computing service. Java Symphony [6] is a Java-centric distributed computing service that aggregates components at runtime (but whose components currently are assumed to be static during the execution).

This paper's purpose is to note *some design issues*, and the *design itself*, for an abstract distributed service component (ADSC)—an abstract class that can be extended to implement concrete distributed service components. We hope that some researchers working on object-oriented distributed computing may value this design discussion since: 1) the design, in our opinion, is not trivial; 2) functional enhancements, refinements, and variations, both small and large, can be described in terms of this base design; 3) implementing the abstract distributed service component or some variant of it in another language is facilitated by an understanding of its design; [1]4) the source code can be downloaded from the JANET web site: http://cs.ucsb.edu/projects/janet/admin.

[1]See, e.g., an implementation of Janet via Corba [9].

2 Janet's Architecture

JANET, a JAva-centric NETwork computing service that supports high-performance parallel computing, is an ongoing project that: virtualizes compute cycles; stores/coordinates *partial* results - supporting fault-tolerance; is partially self-organizing; presumes an open grid services architecture frontend for service discovery (not presented); is largely independent of hardware/OS; is intended to scale from a LAN to the Internet.

JANET comprises an API based on an abstract parallel computational model and a distributed system of service components. The purpose of this architectural overview is to give context to the discussion about the ADSC that follows. JANET is designed to: **support scalable, adaptively parallel computation** (i.e., the computation's organization reduces *completion* time, using many *transient* compute elements, called *hosts*, that may join and leave during a computation's execution, with high system efficiency, regardless of how many hosts join/leave the computation); **tolerate basic faults**: JANET must tolerate host failure and network failure between hosts and other system components; **hide communication latencies**, which may be long, by overlapping communication with computation.

JANET comprises 3 service component classes, which are the *target* of our ADSC class.

Hosting Service Provider (HSP): JANET clients (i.e., processes seeking computation done on their behalf) interact solely with the hosting service provider component. A client logs in, submits computational tasks, requests results, and logs out. When interacting with a client, the hosting service provider thus acts as an agent for the entire network of service components. It also manages the ensemble of service components. For example, when any service component wants to join the distributed service, it first contacts the hosting service provider. If the component is a task server, the HSP tells the task server where it fits in the task server network; if it is a host, the HSP tells the host which task server it is assigned to. When a host fails, its task server notifies the HSP of this fact.

Task Server: This component is a store of Task objects. For the current computation of the current client, each Task object has a representation on some task server, if the task has been spawned, but has not yet been computed. Task servers balance the load of ready tasks among them-

selves. Each task server has a number of hosts assigned to it. When a host requests a task, the task server returns a task that is ready for execution, if any are available. If a host fails, the task server reassigns its tasks to other hosts.

Host: Each host repeatedly requests a task for execution, executes the task, returns the results, and requests another task. It is the central service component for virtualizing compute cycles.

When a client logs in, the HSP propagates that login to all task servers, who in turn propagate it to all their hosts. When a client logs out, the HSP propagates that logout to all task servers, which *aggregate* resource consumption information (execution statistics) for each of their hosts. This information, in turn, is aggregated by the HSP for each task server, and returned to the client. Currently, the task server network topology is a torous. However, scatter/gather operations, such as login and logout, are transmitted via a task server *tree*: a subgraph of the torous. See Figure 1.

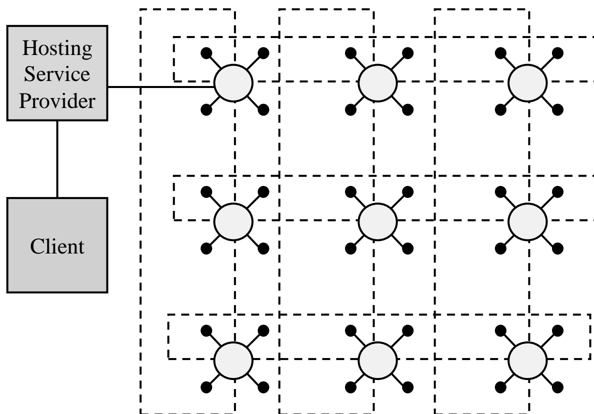

Figure 1. This figure illustrates an instance of the JANET architecture in which there are 9 task servers. The task server topology is a 2D torous (the dashed lines). In the figure, each task server has 4 associated hosts (the little black discs). The client interacts *only* with the HSP.

Task objects *encapsulate* computation: Their inputs & outputs are managed by JANET. Task execution is idempotent, supporting the requirement for host transience and failure recovery. Communication latencies between task servers and hosts are hidden via task caching, task prefetching, and task execution on task servers for selected task classes.

Task servers bind to the service as they become available. Hosts bind to, and unbind from, task servers as they become [un]available, *not* when a client requests service. This enables a rapid response to client service requests, pursuant to JANET's goal of being a high-performance parallel computing service. The time for host [dis]aggregation is not charged to the client; it is managed according to host availability, which is presumed to be transient. This is in contrast to distributed computing approaches where resource aggregation is client-initiated [11]. The difference has to do with a difference of requirements. Ours is to support high-performance parallel computing: The time to respond to a client request must be minimal: We avoid client-initiated aggregation in pursuit of minimal *completion times* for our clients.

To support self-healing, all proxy objects are leased (a basic concept in the Jini architecture). When a task server's lease manager detects an expired host lease and the offer of renewal fails, the host proxy: 1) returns the host's tasks for reassignment, 2) is deleted from the task server.

3 ADSC Architecture

Pursuing the goal of reuse (both within JANET and, possibly, by other distributed computing projects), we formulate an *abstract distributed service component*: A base class for distributed service components. The envisioned abstract class functions as a light-weight, service-to-service component, is *not itself web-based*, but is suitable for subclassing as a Grid service component.

Our view is that a distributed service is a *network* of service components, each of which abstractly is a finite-state machine with output. Such a machine receives commands from its neighbors, and, in response to each command, updates its state and/or sends commands to one or more neighbors. Several design elements derive from this view[2]:

- The remote interface should support a command pattern. This keeps the interface simple and robust: Regardless of how many command classes come into and out of existence (as the service's design evolves), the remote interface between components is stable.
- Each machine maintains a proxy object for each of its neighbors. Neighbor n's proxy is responsible for maintaining the soft state associated with n, and for communicating with n.
- The remote interface provides for remotely invoking commands synchronously and asynchronously (the latter means: insert the command, whose execute method returns void, in a buffer of commands to be sent to the neighbor for invocation).
- In general, *the insertion order of asynchronous commands into an output buffer for a particular neighbor must be preserved in that neighbor's input buffer*: Their communication forms a "channel" (i.e., a queue (aka a FIFO list)). Receiving an asynchronous command out of order may cause the neighbor to update its state incorrectly.

A command channel between neighbors enables us to construct what we call *application-level pipelined multicast*,

[2]Security & privacy are sufficiently complex to require a separate paper in themselves, and thus are omitted.

which forms a virtual command channel between a *source* component and a *set* of distributed components, if those components are connected via a spanning tree of stable components (the interior nodes need to be stable, not the leaves). This communication property may be vital to the correct operation of the distributed service. For example, imagine that one service component, A, needs to send a sequence of commands, (c_1, c_2), to a set, S, of components. Piplelined multicast ensures that, for each component, c_1 is executed before c_2. However, the pipelined nature allows the possibility that component B executes both commands before component D executes either of them—global synchronization is not required. This important functionality follows from a simple induction on the length of the path from component A to each component in set S covered by a spanning tree whose interior nodes are stable. In JANET, such a case occurs, when, for example, the Hosting Service Provider wants to send a client Login command to all Hosts (whose availability is transient) via the spanning tree of task servers (which are presumed to be stable).

A singly-threaded command processor

The architecture of the ADSC is presented as a progression of refinements. The simplest view of an ADSC is that of a singly-threaded command processor: It receives commands; processes them, updating its state, if necessary; sends commands, if necessary, to one or more neighbors. Since we are adhering to the remote object proxy pattern, it typically is the *proxy* that sends the command list to its corresponding remote service component.

Decouple communication from processing

Intuitively, one wants to decouple command processing from communication, which suggests the use of input and output command queues. The thread that executes `receiveCommands()` communicates with the command processor thread via a command queue. Similarly, the command processor thread communicates with the thread that invokes `receiveCommands()` (one for each remote object proxy (i.e., one for each neighbor) via a command queue. This decoupling of communication from computation is depicted in Figure 2.

Piggy-back?

A variation on this theme is shown in Figure 3. Its purpose, called *piggy-backing*, is to reduce thread switching. First, service component A invokes `receiveCommands()` on service component B, passing it a list of commands. Service component B, after inserting these commands into its input queue, sees if it has a nonempty output queue of commands destined for service component A. (This occurs when service component B's send thread for service component A has been notified that there are commands

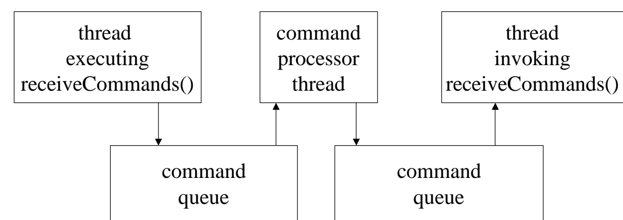

Figure 2. Multi-threading decouples command I/O from processing. The command processor also may update state (not depicted).

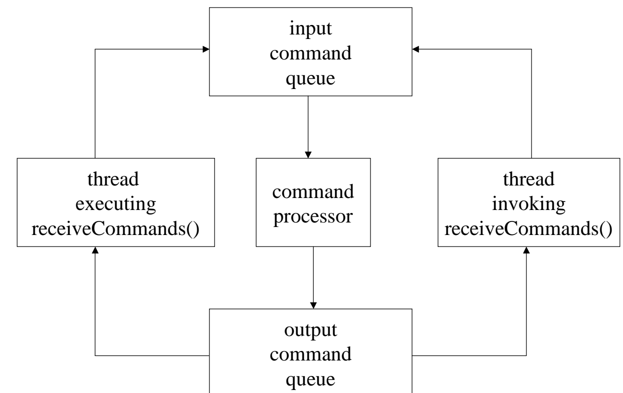

Figure 3. The thread that invokes receiveCommands() also returns commands. The command processor also may update state (not depicted).

to send, but has not yet been scheduled by the JVM.) Service B returns this, possibly empty, list as the return value of `receiveCommands()` that service component A invoked on service component B. Service component A appends this return value to its input queue of commands. The symmetry of this scheme surely has aesthetic appeal.

Using Java RMI in this way *does not preserve the FIFO property* of the command "channel", for 2 reasons:

- The thread on component A that remotely invokes the receive method on component B is not the same as the one that executes the receive method that is remotely invoked by component B. The JVM can swap out these threads so that the order in which the send buffer is emptied is not the order in which the messages go out over the TCP connection. A dual thread swapping problem exists on the receiving component.
- The TCP connection that is used by component A to remotely invoke the receive method on component B is not the same TCP connection that services the receive method that is remotely invoked on component A by component B. Messages sent via separate connections are not guaranteed to be received in the order that they are initiated.

These distinct threads and TCP connections are artifacts of Java RMI, not intrinsic properties of the design. We thus refer to this implementation problem as the *non-FIFO-preserving Java RMI piggy-back anti-pattern*. Moreover, we can work around this problem within the Java RMI framework by sequencing commands between 2 neighbors (i.e., a distinct command sequence for each pair of neighbors). However, the overhead of, in effect, re-implementing the TCP at the application level is too ugly and inefficient to contemplate. In what follows, we abandon piggy-backing; its projected efficiency is not earth-shaking.

Omit needless thread-switching

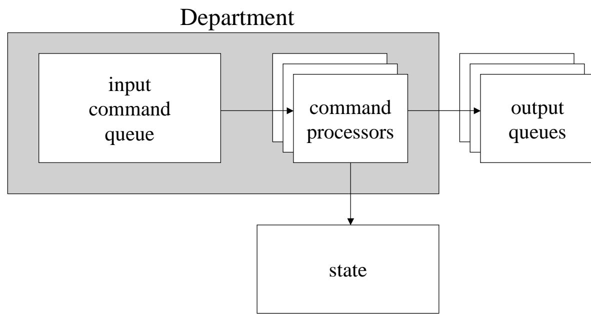

Figure 4. A Department comprises an input command queue and some command processors. These may access/update the containing service component's state and/or produce commands for output to neighbors.

To reduce unnecessary thread-switching, we partition the set of incoming commands into 2 parts: Those that cannot be serviced quickly, and those that can: They require a *small* amount of computation (i.e., it is $O(\log n)$, where n is the size of the state, and is independent of the number of components comprising the distributed service). They never issue a wait(). They never invoke a remote method.

Commands that can be serviced quickly are handled directly by the RMI thread that implements the receive (commands) method: Thread switching is avoided. The other commands must be passed to command processor threads, so that the RMI thread completes quickly. This, in turn, enables the ADSC to support many neighbors. Using a large service organization as a metaphor, we partition the set of "slow" commands, associating each part with an internal *department*. Each department has an input command queue and a *set* of command processor threads (aka workers). (When the service component is executing in a JVM that is running on a machine with multiple CPUs, it may be useful to instantiate multiple command processor threads.) Please see Figure 4. A department object also has access to the state of the service component that contains it; the commands it processes, may read/update this state. The scheme for this abstract service component is depicted in Figure 5.

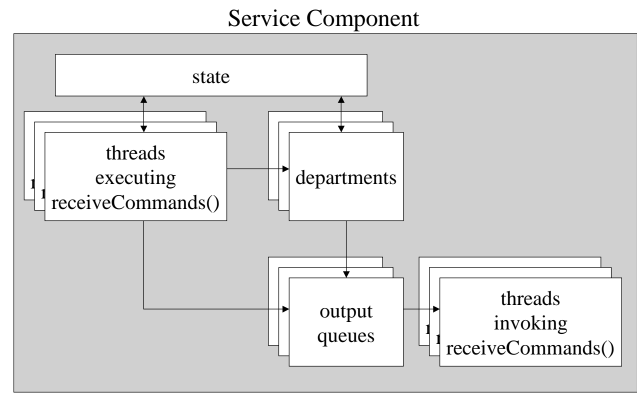

Figure 5. A multi-threaded ADSC architecture.

4 ADSC Design

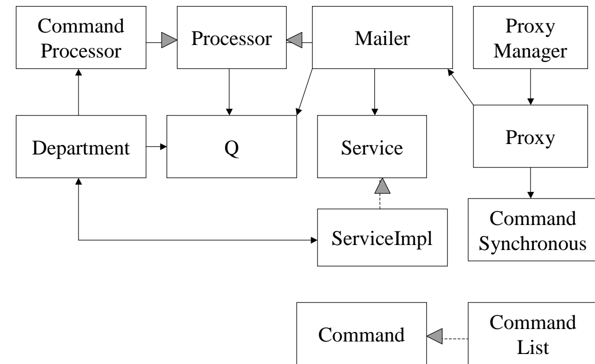

Figure 6. An overview of the ADCS class interactions.

Command classes

All classes of *asynchronous* commands implement the Command interface, which has 2 execute methods:

- An execute method that is invoked by the proxy, to update its representation of the remote component's state, if necessary; process the command locally, if possible; send the command to the remote component, if necessary.
- An execute method that is invoked by the actual remote component. Its argument is a reference to the containing service component, so that the command's execute method can read/update the service component's state. The implementation of this method is empty, if the command is processed entirely by the proxy.

There are times when a service component wants to ensure that a group of commands are sent in one batch. For exam-

ple, when a host finishes executing a Task object, it *usually* insists on sending both a ProcessResult command followed by a RequestTask command. In such cases, the CommandList command is used. Since each Command object in a CommandList may itself be a CommandList, we thus provide a mechanism for sending a *tree* of Command objects. As a tree, these Command objects are disposed of (i.e., either their execute method is invoked, or they are passed to the appropriate department) *in-order* (as opposed to prefix, for example) by the serving component.

When a command must be executed *synchronously* by a remote service component (i.e., the invoking method waits for a return value), the command implements the CommandSynchronous interface, which has 2 execute methods that are similar to asynchronous command. Since, in this case, the proxy does not enqueue the Command object for sending, but rather remotely invokes the executeCommand method, passing the CommandSynchronous object as an argument, the proxy also is passed a RemoteException handler, in case the executeCommand method throws a RemoteException.

The service interface

This remote interface is implemented by all service components. Its 2 remote methods are passed information as to which neighbor component is the client. The methods are:

- `receiveCommands()` which is passed a marshalled List of asynchronous Command objects.
- `executeCommand()` which is passed a CommandSynchronous object whose execute method should be invoked immediately.

The abstract processor class

Processor extends Thread, and has a thread-safe queue of objects to be processed. Its run method removes an object from the queue, and applies the abstract *process* method to it: Currently, it has 2 subclasses:

- CommandProcessor - processes a queue of Command objects (invokes their execute method);
- Mailer - processes a queue of Command lists (remotely invokes `receiveCommands()`, passing a Command list to the Mailer's associated remote service component).

The abstract proxy class

Proxy is responsible for communicating with the its corresponding remote service component. The base class implements 2 execute methods: one for executing Command objects locally (which typically includes appending an *asynchronous* Command object to its Mailer's Command list; the other for remotely invoking an execute method, passing a CommandSynchronous object for immediate invocation by its corresponding remote service component (i.e., the proxy blocks until the remote method invocation returns).

Proxy objects are leased: If there is no activity between a proxy and its corresponding remote service component for an entire lease period, the proxy executes the CommandSynchronous Ping command. If doing so catches a RemoteException, the proxy invokes the abstract `evict()` method (which is implemented by each concrete proxy). Generally, `evict()` performs appropriate clean up (e.g., a HostProxy returns all incomplete Task objects for reassignment), and notifies its containing service that the corresponding remote service component is unavailable (which typically results in the containing service removing the proxy object).

The ProxyManager contains references to a service component's proxy objects (i.e., the proxy objects it has for interacting with its neighbor service components). Thus, it has `add()` and `remove()` methods. It also has a `broadcast()` method, which can be used to send an asynchronous Command object to all neighboring components, via their proxies.

The department class

A service component instantiates 0 or more Department objects. Each such object handles a set of Command classes that are not quickly processed (either they compute intensively, invoke wait(), or invoke a remote method). The Department comprises a queue of Command objects that feeds a Vector of CommmandProcessors. It also has a reference to its containing service component's state, so that CommandProcessors can pass this to the Command objects they process, so that the Command objects can read/update the component's state, if required. For example, JANET's Host service component has a *compute* Department that processes ExecuteTask commands, which are compute intensive. This Department currently instantiates a CommandProcessor for each CPU that is available to the JVM in which the Host is running (e.g., if the machine has 4 CPUs, then 4 CommandProcessor objects are instantiated within the Host's compute Department).

The service implementation class

ServiceImpl is subclassed by every service component. It implements the Service interface, and a `broadcast()` method, for sending its neighbors (via their proxies) an asynchronous Command object. There is an analogous "multicast" method, that "broadcasts" to a Collection of Proxy objects (designating which of the neighbor service components are to receive the asynchronous Command object). It has methods to add/remove proxy objects (managed by its ProxyManager). JANET's principal service components, Hsp, TaskServer, and Host, extend ServiceImpl.

5 Conclusions & Future Work

The ADSC design is based on the abstract view that such a component is a finite-state machine with output that receives commands from neighbor service components, processes those commands, and sends commands to neighbor service components. The design uses the Command pattern and the (Remote) Proxy pattern, and multi-threads command input, processing, and output, while avoiding excessive thread-switching. See http://cs.ucsb.edu/projects/janet for source code. In the future, we intend to address:

- security, based on the Jini Davis project.
- incorporating Jini Extensible Remote Invocation, so the ADSC can select an RMI implementation at runtime (e.g., Ibis's Java RMI [15] or an RMI implementation over a secure transport layer).
- the lack of Domain Name System/IP addressability stemming from dynamically assigned/translated IP addresses, and difficulties with bidirectional communication through NAT units and fire walls.

References

[1] A. Alexandrov, M. Ibel, K. E. Schauser, and C. Scheiman. SuperWeb: Research Issues in Java-Based Global Computing. *Concurrency: Practice and Experience*, 9(6):535–553, June 1997.

[2] A. Bakker, E. Amade, G. Ballintijn, I. Kuz, P. Verkaik, I. van der Wijk, M. van Steen, and A. Tanenbaum. The Globe Distribution Network. In *Proc. 2000 USENIX Annual Conf. (FREENIX Track)*, pages 141–152, San Diego, June 2000.

[3] A. Baratloo, M. Karaul, Z. Kedem, and P. Wyckoff. Charlotte: Metacomputing on the Web. In *Proceedings of the 9th Conference on Parallel and Distributed Computing Systems*, 1996.

[4] P. Cappello and D. Mourloukos. CX: A scalable, robust network for parallel computing. *Scientific Programming*, 10(2):159–172, 2002. Special Issue on Grid Computing, edited by E. Deelman and C. Kesselman.

[5] H. Casanova, G. Obertelli, F. Berman, and R. Wolski. The AppLeS Parameter Sweep Template: User-Level Middleware for the Grid. In *Proceedings of Super Computing*, Nov. 2000. Dallas, TX.

[6] T. Fahringer and A. Jugravu. JavaSymphony: New Directives to Control and Synchronize Locality, Parallelism, and Load Balancing for Cluster and GRID-Computing. In *Proc. ACM Java Grande - ISCOPE Conf.*, pages 8 – 17, Nov. 2002.

[7] J. Frey, T. Tannenbaum, I. Foster, M. Livny, , and S. Tuecke. Condor-G: A Computation Management Agent for Multi- Institutional Grids. In *Proc. Tenth IEEE Symposium on High Performance Distributed Computing (HPDC10)*, Aug. 2000. San Francisco, CA.

[8] A. S. Grimshaw, W. A. Wulf, and the Legion team. The Legion Vision of a Worldwide Virtual Computer. *Communications of the ACM*, 40(1):39–45, Jan. 1997.

[9] R. G. Gutierrez. Implementation of Parallel Algorithms in CORBA and JICOS. Master's thesis, Universidad de Málaga, Spain, June 2003.

[10] K. Kennedy, M. Mazina, J. Mellor-Crummey, K. Cooper, L. Torczon, F. Berman, A. Chien, H. Dail, O. Sievert, D. Angulo, I. Foster, D. Gannon, L. Johnsson, C. Kesselman, R. Aydt, D. Reed, J. Dongarra, S. Vadhiyar, and R. Wolski. Toward a Framework for Preparing and Executing Adaptive Grid Programs. In *Proc. NSF Next Generation Systems Program Workshop (Int. Parallel and Distributed Processing Symp.)*, Apr. 2002. Ft. Lauderdale, FL.

[11] M. Milenkovic, S. H. Robinson, R. C. Knauerhase, D. Barkai, S. Garg, V. Tewari, T. A. Anderson, and M. Bowman. Toward Internet Distributed Computing. *IEEE Computer*, pages 38–46, May 2003.

[12] M. O. Neary and P. Cappello. Internet-Based TSP Computation with Javelin++. In *1st International Workshop on Scalable Web Services (SWS 2000), International Conference on Parallel Processing*, Toronto, Canada, Aug. 2000.

[13] M. O. Neary and P. Cappello. Advanced eager scheduling for Java-based adaptively parallel computing. In *Proc. of the Joint ACM Java Grande - ISCOPE Conference*, pages 56–65, Nov. 2002.

[14] L. F. G. Sarmenta and S. Hirano. Bayanihan: Building and Studying Web-Based Volunteer Computing Systems Using Java. *Future Generation Computer Systems*, 15(5-6):675–686, Oct. 1999.

[15] R. V. van Nieuwpoort, J. Maassen, R. Hofman, T. Kielmann, and H. E. Bal. Ibis: an Efficient Java-based Grid Programming Environment. In *Proc. ACM Java Grande - ISCOPE Conf.*, pages 18 – 27, Nov. 2002.

[16] G. von Laszewski, I. Foster, J. Gawor, W. Smith, and S. Tuecke. CoG Kits: A Bridge between Commodity Distributed Computing and High-Performance Grids. In *ACM Java Grande Conference*, June 2000.

[17] R. Wolski, J. Brevik, C. Krintz, G. Obertelli, N. Spring, and A. Su. Running EveryWare on the Computational Grid. In *Proc. of SC99*, Nov. 1999.