The Component Starting Component: an environment for distributed systems and peer to peer research

Kevin Kane and James C. Browne
Department of Computer Sciences
The University of Texas at Austin
1 University Station C0500
Austin, TX 78712-0233

{kane,browne}@cs.utexas.edu

April 25, 2003

1 Introduction and Motivation

Experimental research on distributed systems needs automated support. Distributed systems, by definition, require processes to start and run on multiple nodes in a network. This is often done either by manually logging into each node and starting processes by hand, or using one of a variety of parallel or grid architectures. These systems require configuration ahead of time so they have an up-to-date list of available workstations. Peer-to-peer systems do not address this problem at all, as they consider the booting and terminating of nodes to be caused by individual users and outside the scope of their control. Peer-to-peer research is inhibited by this lack of control, as simulating joins and leaves in a way that gives reproducible results is necessary for experimentation in this area. Development and debugging of these systems requires several cycles of execution, collection of results, analysis, and then further development to add new functionality and correct programming errors.

A system for supporting experiments in a distributed system needs the following:

- **Discovery of available hosts.** A bootstrapping system must be able to find out what hosts are available to participate in the experiment. In the simplest case, this consists of a static list of hosts which are assumed to be participating. More sophisticated systems use live queries to locate active hosts and prevent staleness of host information.
- **Request authentication.** Any system must ensure that only requests with the proper security credentials are honored. Unauthorized starting of programs on hosts is the main feature of a security breach, and a service whose stated purpose is to start programs is an obvious target for intruders.
- **Filesystem independence.** A distributed system should not assume a shared filesystem. Therefore it is its responsibility to see that binaries are transported to the execution sites when bootstrapping.
- **Host independence.** A distributed system should also not assume homogeneity of its hosts. The system should handle the loading and execution of code on a variety of architectures.
- **Lightweight and easily deployable.** A lightweight system allows for quick, easy setup and deployment of such a system. These should be doable quickly and easily for non-expert users volunteering their hosts for participation. Such a system should also not require any privileged access to the host.
- **Minimal intrusiveness.** A bootstrapping system should minimally impact the host upon which it is deployed, ideally to the point the user is unaware of its impact.
- **Dynamic system structure.** Such a system should allow dynamic structuring of experiments, as these experiments will often involve

joining and leaving protocols, and fault tolerance. A bootstrapping system should allow the experiment to be structured any way without reconfiguration of the bootstrapping system.

We present the "Component Starting Component" (CSC), an environment for launching Java components in a distributed system. Once the CSC is deployed on participating systems in a network, it can stay resident indefinitely. Clients can multicast a solicitation for available hosts to discover CSCs without *a priori* knowledge of their locations. After receiving service offers, clients connect to one or more responding hosts, and then send Java bytecode data and startup instructions. The CSC then starts the component in a different thread in its local Java Virtual Machine (JVM). Communication is guarded by signatures to prevent unauthorized use. The Component Starting Component improves the development and use of distributed systems by providing a straightforward means of moving code to remote sites and starting it both for the incremental development and testing of an application by developers, and the eventual use of the application by end users.

The remainder of the paper is as follows. Section 2 describes the Component Starting Component, its operation, and its deployment. Section 3 analyzes the CSC's fulfillment of the requirements stated above. Section 4 gives related work and contrasts the CSC with similar systems. Section 5 describes two example applications that use the CSC: one in a peer-to-peer setting, and another in a distributed computing setting. Section 6 touches on issues affecting the CSC's performance both in its model and current implementation. Section 7 describes future directions, and section 8 concludes.

2 Component Starting Component Protocol

The Component Starting Component is deployed on individual hosts, and accepts requests to start components on the local host. It operates in a single multicast domain; that is the expanse of network from the point of origin through which multicast datagrams will be routed. This forms a "virtual organization" of component hosts in which to deploy a distributed or peer-to-peer application. A virtual organization is one whose members are geographically apart, but working together so as to appear as a single, unified organization [20]. Once the CSC is deployed, it makes

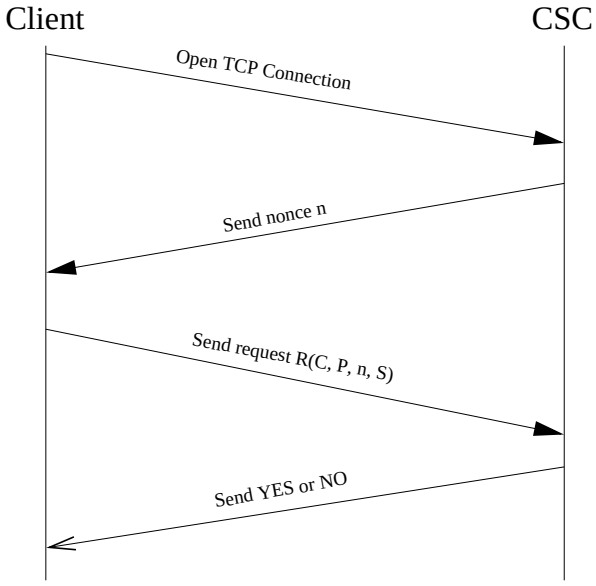

Figure 1: Component Starting Component Protocol

conducting experiments easy. It can remain permanently resident on hosts and be used in a number of different experiments. Each operating CSC and any clients all share a Digital Signature Algorithm public/private key pair. The secure dissemination of this keypair is beyond the scope of the system.

An application running on a client formulates a request to start a number of components. A client first solicits, via a reliable multicast, all available Component Starting Components. Currently, the reliable multicast is implemented by the Light-weight Reliable Multicast Protocol (LRMP) [23]. It then waits a user-defined time to receive "offers," the endpoint addresses of all operating CSCs. Assuming it receives at least one solicitation, it iterates over its list of requests, and iterates over the list of available CSCs, and dispatches a request to each in turn, looping around the CSC list as many times as necessary so as to most evenly spread the components across available hosts.

Pseudocode for the CSC is as follows. C represents communication parameters. These consist of the multicast group (denoted $C.g$), a port number (denoted $C.p$) to use, and a multicast time-to-live (denoted $C.t$). K_{pub}, K_{priv} are the DSA public/private key pair. H represents the address of the local host. $R(B, P, n, S)$ is a client request, where B is a code bundle, P specifies the method and arguments to invoke, n is a nonce, and S is the signature over all the other fields.

```
CSC(C, K_pub)
    start CSC-Multicast-Listener(C) in its own thread
    start CSC-TCP-Listener(C, K_pub, K_priv) in its own thread
end

CSC-Multicast-Listener(C)
    m ← open multicast socket (C.g, C.p)
    while (true)
        xid ← m.receive()
        m.send(xid, H, C.p)
end

CSC-TCP-Listener(C, K_pub)
    l ← open point-to-point listen socket (C.p)
    while (true)
        s ← l.accept()
        n ← random
        s.send(n)
        R(B, P, n, S) ← s.receive()
        if (verify_signature(R, n, K_pub))
          s.send(YES)
          load code bundle R.B into Java VM
          launch another thread according to R.P
        else
          s.send(NO)
        s.close()
end
```

Pseudocode for the client is as follows. C specifies communication parameters as above, and $Q = R_0, \ldots, R_n$ is an array of requests to make. Each R_i contains a code bundle B and starting parameters P. K_{priv} is the private key corresponding to K_{pub} above. m.receive(t), where m is a multicast socket, represents receiving on the multicast socket until a timeout interval t has elapsed.

```
Make-CSC-Requests(C, Q, K_priv, t)
    m ← open multicast socket (C.g, C.p)
    xid ← random
    { Solicit available CSCs }
    m.send(xid)
    offers ← m.receive(t)
    for each R_i ∈ Q
        for each O ∈ offers
            s ← connect(O)
            if s is null
              offers ← offers − O
              try again with next O ∈ offers
            n ← s.receive()
            S ← signature(R_i.B, R_i.P, n, K_priv)
            create request R(R_i.B, R_i.P, n, S)
            s.send(R)
            r ← s.receive()
            if r = YES
              proceed to next request
            else try again with next O ∈ offers
end
```

The interaction between a client and an individual CSC proceeds as in figure 1.

2.1 CSC Deployment

The Component Starting Component must first be started on the hosts to be used in the distributed systems. This procedure is intentionally simple to make deployment quick and inobstrusive. The procedure is:

1. Download CSC Java Archive (JAR) file to host.
2. Securely transfer security keys to host.
3. Execute CSC with security keys and communication parameters.

The secure transfer of the keys is done out-of-band through a mechanism such as Secure Shell (SSH). These three steps can be automated with a script.

3 Analysis of Requirements

The Component Starting Component uses a multicast to discover all available hosts at the moment components are ready to start. This requires no advance knowledge of the hosts that are available, only the multicast group being used. Presently it is limited to use within a multicast domain, but a network of CSCs could be used to bootstrap systems using other communications layers, such as the planned implementation of the CSC atop Scribe [10], a peer-to-peer group communication package based on Pastry [9]. This would allow virtual organizations to span several multicast domains.

All requests must be signed by the provided security keys. Any unsigned or improperly signed requests will be rejected, which protects against requests being altered in transit.

Requests also contain all the necessary code to start components. The only software required on each site is the Java system, and the CSC code itself. No sharing of filesystems or copying of files is otherwise required. Users deploying the CSC must

have accounts on the hosts, but once the CSC is deployed, any user, even those without accounts, can make use of the CSC when they posess the proper security keys. At present the CSC will run components with the full privileges of the virtual machine. Future versions will use Java's permissions architecture as described in section 7.

The CSC works with Java-based components, and therefore leverages Java's platform independence. Any components are therefore applicable to any host on which a Java environment exists.

The CSC is also lightweight, and easily deployable. A three-step process is all that is required to deploy the CSC, and the system remains idle when not in use. It therefore proides minimal impact on the system. At present there is no way to control resource consumption of Java applications aside from limitations imposed by the operating system on the entire virtual machine itself, but such resource management is planned for Java's future [19].

Finally, the CSC allows components to be dynamically started at any time, and its discovery mechanism allows these components to be spread across whatever hosts are available at the time of the request. This allows as many copies to be started as desired for fault tolerance and replication, and can be done so without regard to which specific hosts are online at the time.

4 Related Work

Remote procedure call (RPC) type systems, such as Grid RPC, Java RMI, and Web Services all provide for the remote invocation of programs. In all these cases, components must already be executing on those remote sites and have registered their interfaces. Grid RPC provides some discovery mechanism by maintaining mappings between grid-enabled services and the actual hosts where they reside. In all cases, these services must be started ahead of time by some out-of-band means; any facilities for starting up new services must be provided by such a service rather than by the RPC mechanism itself.

Simple distributed systems use manual startup of components, or use a combination of shell scripts and remote login utilities like `rsh` and Secure Shell. These approaches are typically non-scalable, and offer no dynamic discovery of available hosts, as the hosts to be used are manually coded into the shell script. This approach also delegates authentication and authorization to the remote login utility. Often, these approaches also assume a shared filesystem, and those that don't must use a file copying utility, such as rcp or scp. Any allowances for host heterogeneity must also be manually handled by the system, or the hosts must be kept homogenous.

The Message Passing Interface (MPI) [2] and Parallel Virtual Machine (PVM) [3] both utilize daemons running on each participating host to start new processes. However, each refers to a static administrator-provided listing of available hosts to discover where processes can be started, and provide no automatic discovery. Both systems relegate the details of authentication and authorization to other system utilities, such as *remote shell* (`rsh`) or *secure shell* (`ssh`). Similar utilities are also used to copy program binaries to execution sites. Java Parallel Virtual Machine (jPVM) [4], a system based on the ideas of PVM implemented in Java, requires that after the daemons are started on individual hosts, the user then constructs a list of their execution locations so that daemons may find each other. No authentication is performed to guard access to spawning new processes. All executed code must also already be resident on the remote host; jPVM supports no mobile code. jPVM is a "lightweight" system, but starts new instances of the Java Virtual Machine for each component booted.

Globus [5] uses a certificate system where authenticated users have a certificate signed by a trusted Certifying Authority (CA). These credentials are then authorized by a particular site. This guards access to the resources, but job dispatching is still done through a statically provided list of available hosts. Globus is also a "heavyweight" system, where the overhead of installation and maintenance is high.

Legion [7], like Globus, also provides a security architecture for guarding access to resources, as well as a unified naming model for access across its data grid using public and private keys. Available hosts for execution are maintained by objects existing in Legion's distributed filesystem, where a host or set of hosts manifests itself as an abstract Host object, exporting an interface for execution of programs [8]. Therefore, if a host is aware *a priori* of the existence of a Legion data grid, it can make itself available to the grid for computation. Code bundles are not sent explicitly with these requests; rather code is placed in Legion's distributed file system and referenced in the request. Legion is also a "heavyweight" system.

Klava [1] is a distributed tuple space system based on many of the ideas of Linda [6]. Klava implements

dynamic starting of execution engines that it calls "Nodes." These nodes are coordinated through a "Net," which is a centralized coordination point that maintains a list of all operating nodes. Klava supports sending bundles of code to remote sites for execution and for starting new Nodes, although there is no mention of security protections around these functions. But since Klava uses the Linda model, it is less suited to distributed systems research. Features useful for distributed systems research, such as associative matching, must be manually implemented atop Linda primitives.

Web services [12] are the most popular manifestation of component-based architectures. Using the universal XML [13] standard allows communication between different architectures. The overhead of setting up a "servlet container" can be initially heavy, but deployment of web services themselves once the container is operational is simple and straightforward; even very simple Java classes can be converted into a web service with little or no change at all. Web services can be discovered through centralized indexes such as UDDI [14]. Web services are tied to their hosts, however, as servlet containers load their services from files resident on the host system. There is no provison for loading a web service from the network, and no provison for the dynamic restructuring of a service at runtime, apart from deactivating the service and booting a new one in its place.

The "Grid MetaProcessor" [17] extends the computation model of SETI@home [18] by introducing an agent that includes a kernel which allows the pulling down of software and data for processing in the background, utilizing idle periods of the system just as SETI@home does. Under this model, clients request units of work when they are ready, as opposed to started at the behest of the application.

PlanetLab [21] is an international set of hosts made available by component research institutions for wide-area network research, which is not unified by any multicast domain. It uses a resource reservation system called "slices" [22] to control use of component hosts. SliceDeploy by Robert Adams, available from the PlanetLab web site, is an application which does the job of shell scripts to make reservations in this system, copy programs to component hosts, and start or stop them at will. This system is lightweight and takes care of filesystem independence, while using system utilities to handle request authentication. SliceDeploy provides no discovery features, and requires all component hosts to be manually specified on the command line.

5 Example Applications

5.1 Experiments with Peer-to-Peer Systems

Most research on establishment and use of peer-to-peer networks and their applications use ad-hoc or uncontrolled environments for their execution environments. Systematic evaluation of peer-to-peer algorithms and applications requires establishment of known and reproducible system configurations and execution environments. This section describes the use of the CSC to instantiate Pastry [9] and Scribe [10] package built atop it. Scribe is a group communication package built on top of Pastry, a peer-to-peer distributed hash table system. FreePastry [11] is an implementation of these and other sample programs built atop Pastry, and is available in Java as a Java Archive (JAR) file. Scribe comes with a simple test application, which starts a default of 5 nodes on the local host, which form several groups and publish a message every ten seconds. These hosts may themselves exist in separate multicast domains, although at present this implies multiple networks of CSC, each occupying a single multicast domain.

A single node is started in a well-known location manually. Pastry applications require a "bootstrap" node in a well-known location. Any number of other nodes are then started with the CSC, including a parameter that specifies the location of the "bootstrap" node. This instantly creates a peer-to-peer network with any desired number of physical nodes, each with five virtual nodes resident.

The "bootstrap" node is manually started from the command line on a well known location. A second program, which makes CSC requests, is then started. This program will construct a number of those requests, which will include the name of the "bootstrap" location as part of the starting parameters, and make those requests following the protocol specified in section 2.

This allows for the simulation of a network of communicating entities, which can be used for a variety of applications. A distributed system that makes heavy use of group communication, such as CoorSet below, could use such a network as its communications layer instead of a traditional multicast. Large networks of data use distributed hash table systems like Pastry, and this system can be used to bring up a suitably-

sized network before populating it with data.

5.2 Experiments on Distributed Algorithms and Computations

Systematic experimental evaluations of distributed algorithms and distributed computations have been difficult to conduct for the same reasons as for peer to peer systems, which are a special case of the more general concepts of distributed systems. We have implemented and are applying a distributed programming system based on broadcast communication [16]. The first motivation for providing support for implementation of experimental research on distributed systems was the effort required to manually conduct experiments evaluating the properties and behaviors of algorithms programmed in the CoorSet programming model.

The CoorSet [16] system, short for **Coor**dinating **Set**, is an experimental component-based distributed architecture, where components are specified in terms of their exported interfaces. The initial configuration of the distributed application is specified in a single configuration file, which specifies the interfaces and numbers of each component to launch.

Each component maintains a *profile*, which is a list of attribute/value pairs that describe a component's state. All communication is *broadcast communication*, where messages are sent to all components. Components also maintain an *accepts interface* which specifies message types it can receive, and a *requests interface* which specifies message types it will need to send. Messages are addressed using *selectors*, which are boolean logic propositions specifying properties of the receiver set. Each component evalutes the selector against its local profile, and if the selector evaluates true, the component is a member of the receiver set and processes the message. This allows runtime matching of interfaces with any components that are online at the time a request is made.

CoorSet requires the ability to launch components on nodes around the network when the application is started. It does this by using the Component Starting Component, using the user's definition of an individual component to gather the required binaries, bundle them up in a CSC request, and dispatch them to available CSCs. This allows it to distribute all the components of an application across all available hosts. Without this functionality, a user would need to manually launch the initial components in the application by hand at various locations in the network.

5.2.1 CoorSet Example

Consider a distributed readers and writers system in CoorSet. Most of the details of this system are beyond the scope of this paper, but we give an overview as an illustration. The system configuration file specifies a set of active data objects which implement a replicated data item. The configuration file specifies their initial external interface, which they use to maintain consistency amongst each other, and to service reading and writing requests from clients.

Such a system consists of k replicas of a single data item. In a more general case there would be multiple data items, but for this example we restrict ourselves to a single datum. Each replica will identify itself by its index $i, 0 \leq i \leq k-1$. In a production deployment, clients will come up and interact on their own, but in an experimental deployment, a number r readers of various data items will be instantiated as well. The components will be deployed as follows:

1. CoorSet formulates a list of $k+r$ CSC requests R, one for each of the k replicas to service, and one for each of the r experimental readers.

2. The CSC client receives this list, and makes a multicast solicitation for available CSCs.

3. The client then waits a time t while it receives a number of offers from available CSCs.

4. With the list of received offers O and requests R, the client then connects to each CSC in O in a round-robin fashion, submitting an individual request in R. As each request is accepted by a CSC, it is removed from the list R. When R is empty, the client completes.

5.3 Example of Development Cycle

The previous two examples show instances of already completed applications being deployed in a production circumstance. The Component Starting Component is also useful for the development of a distributed application, as it saves the repeated steps to log into remote hosts, copy a new revision of a component, and then launch it. This cuts down on the develop-test-analyze cycle of software development.

Consider an implementation of the classic "Dining Philosophers" problem. There are n philosophers sitting around a table, n forks on the table, one on each side of each philosopher, and $n-1$ meal tickets in the middle of the table (to prevent deadlock).

Each philosopher, fork, and meal ticket exists as an entity in the distributed system. Each fork and meal ticket does not initiate any activity, but merely responds to requests to be picked up or put down, or to terminate.

Each philosopher does the following:

1. Obtain a meal ticket.
2. Obtain the fork on either side of the philosopher. These will be forks numbered i and $i + 1 \bmod n$ for philosopher i.
3. "Eat." In this implementation, this just represents a successful completion of one loop.
4. Return both forks and the meal ticket to the table.

This process is repeated a configurable number of times. But even for a modest system of 5 philosophers, this requires 5 philosopher components, 5 fork components, and 4 meal ticket components to be running in the system, which would be a considerable amount of work to do by hand for each iteration of the development cycle. The CSC allows the code to be developed for each of these three types of components, and then deployed from a single centralized location, their output observed and then used in subsequent analysis for future iterations. The work of logging in to fourteen different systems and relaunching each component is saved.

6 Performance Analysis and Limitations

In this section we discuss some of the performance issues of the Component Starting Component. We focus on the communication issues between clients and candidate hosts as that will dominate its performance.

The central issue to its performance is the multicast used in the initial solicitation. Since no central list of active CSCs is maintained, a client must "guess" an amount of time to wait for offers to arrive. This amount of time is highly dependent on the scope of the multicast domain and the level of connectedness amongst the candidate hosts. On a single network with well-connected hosts, offers arrive almost immediately and from all candidate hosts. Evaluation on a larger multicast domain that spans multiple networks was not conducted, due to no such domain being available at the time of this writing. Still, in an unknown network situation, an out-of-band means must be used to configure the delay the client waits before starting to make requests. Offers may come in after the client begins to distribute components, and when they do they will be added to the list, but this can cause an uneven distribution of work.

When a work unit is dispatched to a remote host, the request is sent over a point-to-point reliable TCP connection. If the same code bundle is to be used in more than one requests, the code will be sent out over the wire multiple times. Large code bundles are unsuitable for the fixed-length datagrams used in multicasting, and the current protocol, LRMP, enforces a Maximum Transfer Unit size of 1400 data bytes per datagram.

Each multicasted solicitation carries with it a random transaction identifier on the interval $[0, 800000)$. These random identifiers are generated from Java's `java.util.Random` class. This class bases its seed on the current time, so requests made when the local clock contains the same value as the local clock of another client when it makes its request, they will use the same random transaction identifiers. In this case, each will also receive offers resulting from the other's solicitation. This unlikely possibility was deemed acceptable as opposed to using a more computationally-intensive cryptographic random number generator, such as Java's `java.security.SecureRandom` class.

7 Future Work

A necessary step to move beyond the limiting box of IP Multicast is to move to an overlay network type multicast, like Scribe [10] or End System Multicast [15]. IP Multicast suffers from administrative limitations which are unlikely to be removed, making it unsuitable for communication beyond a single administrative domain. A multicast mechanism which can span multiple networks allows for even larger distributed systems. This will then make the CSC applicable to PlanetLab, where the hosts are geographically distributed and exist in different multicast domains.

Currently, all clients and servers share the same public/private DSA key pair. Use of X.509 certificates with authorized signatures directly extends this authentication mechanism into one where authority is more easily delegated, and also provides the possibility of Certificate Revocation Lists, allowing such privileges to be revoked.

At present, the CSC starts components as simple threads in its own virtual machine, with no security protection at all. Although this will not interfere with the correct operation of benign software, this does give any software started total access to the virtual machine, which could allow it to interfere with other threads or terminate the virtual machine, CSC and all. Java provides an elaborate access control mechanism for controlling applications' access to the virtual machine, although it is disabled by default when running normal applications. Use of this permissions system allows each site to configure permissions for each application if desired, and specify a minimum set of permissions for all such applications. The CSC also does not yet keep track of the components it has started, and which are running and which have terminated, but such a function and an interface for querying it can be useful for monitoring components both for the development cycle, and for production use.

Many distributed systems currently exist only as native code, so future work also includes the transport and dynamic loading of shared libraries that make use of the Java Native Interface (JNI). By definition these libraries will be platform-specific, and will therefore require facilities to match the appropriate libraries to the corresponding architectures. But this can be used to escape Java for those applications which still exist as native code. In a future version of the CSC, a native code library can be transmitted as part of the code bundle, dynamically loaded into the virtual machine, and then invoked from a stub Java class.

For applications which are completely independent of Java, the CSC could be used just as a transport mechanism. The same means can be used to transmit code bundles, but they can be then written to local disk and passed to the underlying operating system for execution. The CSC would then be outfitted with information about its local host, and would only respond to solicitations for its particular platform.

8 Conclusion

Distributed systems of all kinds, from traditional master-slave architectures to current peer-to-peer methods, all require some method of supporting experimental research in an automated and reproducible way, and in the case of some architectures, for production use. We have provided such an environment for systems based in Java, that can be bundled up and sent across the network to participating hosts, with proper authentication protecting those hosts from unauthorized users attempting to run their own code. This allows applications to be tested during development by simplifying the process of execution for testing purposes, and once an application is ready for production use, it can be deployed across a network in the same way.

References

[1] Lorenzo Bettini, Rocco De Nicola, and Rosario Pugliese. "Klava: a Java Framework for Distributed and Mobile Applications." *Software – Practice and Experience*, 32:1365-1394, 2002.

[2] The Message Passing Interface (MPI). http://www-unix.mcs.anl.gov/mpi/

[3] Parallel Virtual Machine (PVM). http://www.csm.ornl.gov/pvm/pvm_home.html

[4] Adam J. Ferrari. "JPVM: Network Parallel Computing in Java." *Concurrency: Practice and Experience*, 10(11-13):985-992, 1998.

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

[6] David Gelernter. "Generative communication in Linda." *ACM Transactions on Programming Languages and Systems*, 7(1):80-112, 1985.

[7] A. S. Grimshaw, A. J. Ferrari, and E. West. "The Legion Vision of a Worldwide Virtual Computer." *Communications of the ACM*, 40(1):39-45, January 1997.

[8] Andrew S. Grimshaw, Michael J. Lewis, Adam J. Ferrari, and John F. Karpovich. "Architectural Support for Extensibility and Autonomy in Wide-Area Distributed Object Systems." University of Virginia Technical Report CS-98-12.

[9] A. Rowstron and P. Druschel, "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems". *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, Heidelberg, Germany, pp. 329-350, November, 2001.

[10] A. Rowstron, A-M. Kermarrec, M. Castro and P. Druschel, "SCRIBE: The design of a large-scale event notification infrastructure", *NGC2001*, UCL, London, November 2001.

[11] Peter Druschel, Eric Engineer, Romer Gil, Y. Charlie Hu, Sitaram Iyer, Andrew Ladd, et. al. "FreePastry." Software available at http://www.cs.rice.edu/CS/Systems/Pastry /FreePastry/.

[12] World Wide Web Consortium. "Web Services." http://www.w3.org/2002/ws/.

[13] World Wide Web Consortium. "Extensible Markup Language (XML)". http://www.w3.org/XML/.

[14] "Universal Description, Discovery, and Integration of Web Services (UDDI)". http://www.uddi.org/.

[15] Yang-hua Chu, Sanjay G. Rao, and Hui Zhang. "A Case for End System Multicast." *ACM SIGMETRICS*, pp. 1-12, June 2000.

[16] Kevin Kane. "CoorSet: An Interface Description Language for Associatively-Coordinated Components." *Preprint.*

[17] United Devices. "Grid MetaProcessor." http://www.ud.com/.

[18] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. "SETI@home: An Experiment in Public-Resource Computing." *Communications of the ACM*, 45(11):56-61, November 2002.

[19] Li Gong. "Java 2 Platform Security Architecture." http://java.sun.com/j2se/1.4.1/docs/guide/security /spec/security-spec.doc.html.

[20] TechTarget Network. "Whatis virtual organization."

http://whatis.techtarget.com/

[21] Larry Peterson, Tom Anderson, David Culler, and Timothy Roscoe. "A Blueprint for Introducing Disruptive Technology Into the Internet." *Proceedings of the 1st Workshop on Hot Topics in Networks*, October 2002.

[22] The PlanetLab Architecture Team. "Dynamic Slice Creation." PlanetLab Design Notes PDN-02-005. http://www.planet-lab.org/pdn/pdn02-005.pdf.

[23] Tie Liao. "Light-weight Reliable Multicast Protocol." http://webcanal.inria.fr/lrmp/lrmp_paper.ps

[24] J. C. Browne, K. Kane, and H. Tian. "An Associative Broadcast Based Coordination Model for Distributed Processes." *Coordination Models and Languages, Proceedings of COORDINATION 2002*, Lecture Notes in Computer Science 2315, Springer-Verlag, p. 96.