CX: A Scalable, Robust Network for Parallel Computing

Peter Cappello and Dimitris Mourloukos
Computer Science Dept.
University of California
Santa Barbara, CA 93106
{cappello | mourlouk}@cs.ucsb.edu

Abstract

CX, a network-based **c**omputational e**x**change, is presented. The system's design integrates variations of ideas from other researchers, such as work stealing, non-blocking tasks, eager scheduling, and space-based coordination. The object-oriented API is simple, compact, and cleanly separates application logic from the logic that supports interprocess communication and fault tolerance. Computations, of course, run to completion in the presence of computational hosts that join and leave the ongoing computation. Such hosts, or producers, use task caching and prefetching to overlap computation with interprocessor communication. To break a potential task server bottleneck, a network of task servers is presented. Even though task servers are envisioned as reliable, the self-organizing, scalable network of n servers, described as a *sibling-connected height-balanced fat tree*, tolerates a sequence of $n-1$ server failures. Tasks are distributed throughout the server network via a simple "diffusion" process.

CX is intended as a test bed for research on automated silent auctions, reputation services, authentication services, and bonding services. CX also provides a test bed for algorithm research into network-based parallel computation.

1 Introduction

The ocean contains many tons of gold. But, the gold atoms are too diffuse to extract usefully. Idle cycles on the Internet, like gold atoms in the ocean, seem too diffuse to extract usefully. If we could harness effectively the vast quantities of idle cycles, we could greatly accelerate our acquisition of scientific knowledge, successfully undertake grand challenge computations, and reap the rewards in physics, chemistry, bioinformatics, and medicine, among other fields of knowledge.

Several trends, when combined, point to an opportunity:

- The number of networked computing devices is increasing: Computation is getting faster and cheaper: The number of unused cycles per second is growing rapidly
- Bandwidth is increasing and getting cheaper
- Communication latency is *not* decreasing
- Humans are getting *neither* faster *nor* cheaper.

These trends and other technological advances lead to opportunities whose surface we have barely scratched. It now is technically feasible to undertake "Internet computations" that are technically *infeasible* for a network of supercomputers in the same time frame. The maximum feasible problem size for "Internet computations" is growing more rapidly than that for supercomputer networks. The SETI@home project discloses an emerging global computational organism, bringing "life" to Sun Microsystem's phrase "The network is the computer". The underlying concept holds the promise of a huge computational capacity, in which users pay only for the computational capacity actually used, increasing the utilization of existing computers.

1.1 Project Goals

In the CX project, we are designing an open, extensible **C**omputation e**X**change that can be instantiated privately, within a single organization (e.g., a university, distributed set of researchers, or corporation), or publicly as part of a market in computation, including charitable computations (e.g., AIDS or cancer research, SETI). Application-specific computation services constitute one kind of extension, in which computational consumers directly contact specialized computational producers, which provide computational support for particular applications.

The system must enable application programmers to design, implement, and deploy large computations, using computers on the Internet. It must reduce human administrative costs, such as costs associated with:

- downloading and executing a program on heterogeneous sets of machines and operating systems
- distributing software component upgrades.

It should reduce application design costs by:

- giving the application programmer a simple but general programming abstraction
- freeing the application programmer from concerns of interprocessor communication and fault tolerance.

System performance must scale both up and down, despite communication latency, to a set of computation producers whose size varies widely even within the execution of a single computation. It must serve several consumers concurrently, associating different consumers with different priorities. It should support computations of widely varying lifetimes, from a few minutes to several months. Producers must be secure from the code they execute. Discriminating among consumers is supported, both for security and privacy, and for prioritizing the allocation of resources, such as compute producers.

After initial installation of system software, no human intervention is required to upgrade those components. The computational model must enable general task decomposition and composition. The API must be simple but general. Communication and fault tolerance must be transparent to the user. Producers' interests must be aligned with their consumer's interests: computations are completed according to how highly they are valued.

1.2 Some Fundamental Issues

It is a challenge to achieve the goals of this system with respect to performance, correctness, ease of use, incentive to participate, security, and privacy. Although this paper does not focus on security and privacy, the Java security model [15] and the "Davis" release of Jini address network security [21] (covering authentication, confidentiality, and integrity) clearly are intended to support such concerns. Our choice of the Java programming system and Jini reflects these benefits implicitly.

In this paper, we present the *Production Network* service subsystem of CX, focusing on its design with respect to application programming complexity, administrative complexity, and performance. Application programming complexity is managed by presenting the programmer with a simple, compact, general API, briefly presented in the next section. Administrative complexity is managed by using the Java programming system: Its virtual machine provides a homogeneous platform on top of otherwise heterogeneous sets of machines and operating systems. We use a small set of interrelated Jini clients and services to further simplify the administration of system components, such as the distribution of software component upgrades. The Production Network is a Jini service that interfaces with every other CX Jini client and service. We however focus in this paper on the Task Server (a Jini service) and the Producer and Consumer, which are Jini clients.

Performance issues can be decomposed into several sub-issues.

Heterogeneity of machines/OS: The goal is to overcome the administrative complexity associated with multiple hardware platforms and operating systems, incurring an acceptable loss of execution performance. The tradeoff is between the efficiency of native machine code vs. the universality of virtual machine code. For the applications targeted (not, e.g., real-time applications) the benefits of Java JITs reduce the benefits of native machine code: Java wins by reducing application programming complexity and administrative complexity, whose costs are not declining as fast as execution times.

Communication latency: There is little reason to believe that technological advances will significantly decrease communication latency. Hiding latency, to the extent that it is possible, thus is central to our design.

Scalability: The architecture must scale to a higher degree than existing multiprocessor architectures, such as workstation clusters. Login privileges must not be required for the consumer to use a machine; such an administrative requirement limits scalability.

Robustness: An architecture that scales to thousands of computational producers must tolerate faults, particularly when participating machines, in addition to failing, can disengage from an ongoing computation.

1.2.1 Ease of use

The computation consumer distributes code/data to a heterogeneous set of machines/OSs. This motivates using a *virtual* machine, in particular, the JVM. Computational producers must download/install/upgrade system software (not just application code). Use of a screensaver/daemon obviates the need for human administration beyond the one-time installation of producer software. The screensaver/daemon is a wrapper for a Jini client, which downloads a "task server" service proxy every time it starts, automatically distributing system software upgrades.

1.3 Paper Organization

In the next section, we discuss related work, particularly noting those ideas of others that we have incorporated into CX. In section 3, we introduce the API. In section 4, we describe CX's architecture. In section 5, we present results from preliminary experiments. The Conclusion summarizes our contributions and some directions for future work.

2 Related Work

Legion [16] and Condor [11] were early successes in network computing. They predate Java, hence are not Java-centric, and indeed do not use a virtual machine to overcome the portability/interoperability problem associated with heterogeneous machines and OSs. The use of a virtual machine is a significant difference between Java-centric and previous systems. Java-centric systems, among other differences, do not require computational consumers to have login privileges on host machines. Charlotte [4] was the first research project, to our knowledge, that was Java-centric. Charlotte used eager scheduling, introduced by the Charlotte team, and implemented a full distributed shared memory. Cilk-NOW [6], based on Cilk 2, provides for "well-structured" computations (a strict subset of dag-structured computations). It uses work-stealing and checkpointing (to a shared filesystem, such as NFS) for adaptively parallel computations (i.e., computations hosted by machines that may join/retreat from the computation dynamically [8]). Atlas [3], a version of Cilk-NOW intended for the Internet setting, put three important concepts into a Java-centric package: a computational model that supports well-structured computations, work-stealing, and the Internet. It was an attempt to create a Java-centric parallel processor with machines on the Internet. As a masters project, it terminated abruptly, and, in our opinion, without reaching its full potential. CX shares these three properties. However, we have discovered that the dag-structured task model, eager scheduling (instead of Atlas's checkpointing), work-stealing, and space-based coordination integrate so elegantly as to be "made for each other" when an adaptively parallel computation is deployed on a network. Globus [13] is a metacomputing or umbrella project. It consequently is not Java-centric, and indeed *must be* language-neutral. CX is intended to fit under Globus's umbrella via a *portal* [22]. Javelin [19, 20, 9] is Java-centric, implements work stealing and eager scheduling, and has a host/broker/client architecture. Javelin's implementation of eager scheduling is centralized on the client process.

Recently, several systems have emerged for *distributed* computations on the Internet. Wendelborn et al. [24] describe an ongoing project to develop a geographical information system (PAGIS) for defining and implementing processing networks on diverse computational and data resources. Hawick et al. [17] describe an environment for service-based meta-computing (DISCWorld). Fink et al. [12] describe Amica, a meta-computing system to support the development of coarse grained location-transparent applications for distributed systems on the Internet, and includes a memory subsystem. Bakker et al. [2] take the view of distributed objects as their unifying paradigm for building large-scale wide area distributed systems. They appear to intend to do for objects what the world wide web did for documents. Objects can differ on their scheme, if any, for partitioning, replication, consistency, and fault tolerance, in a way that is opaque to clients.

Huberman et al. [1] relate anonymity to incentives, in their application of the "tragedy of the commons" to anonymous peer-to-peer networks.

Securing the Jini infrastructure is not the focus of this project; commercial efforts are under way to do this.

Recent commercial ventures attest to the perception that unused cycles can be made available in a computationally meaningful way. Such ventures, while still in their infancy, include EnFuzion (targeted at intranets), Applied Metacomputing (the commercialization of Legion), Distributed Science (aka the ProcessTree), Entropia, Parabon Computation, Popular Power, and United Devices.

The setting for CX is the Internet (or an intranet). It comprises a set of interrelated Jini services and clients implemented in Java. From a performance point of view, its goal is somewhat different from both the commercial ventures and the early systems such as Legion and Condor. These systems are intended primarily to increase system throughput or utilization of idle cycles. CX is intended to *push the limits of parallel computing in a network setting*, despite long communication latencies. Its architecture incorporates ideas from a variety of sources. Briefly, it uses thread programming model ideas from Cilk [5]; scheduling ideas from Enterprise [18], Spawn [23], and Cilk; classic decoupled communication ideas from Linda [8] (and JavaSpaces [14], its Java incarnation); eager scheduling ideas for fault tolerance from Charlotte; and the host/broker/client architectural ideas from Javelin. To match supply with demand [7] in time and space, the system incorporates the concept of auctions [10] via a market maker.

This article outlines the rationale for these choices, as they pertain to the design of CX's ProductionNetwork subsystem.

3 API

Computational Model

The *computational model* reflects the dominating physical constraint on networked computation among compute producers whose availability may be short-lived: long communication latency relative to execution speed. Computation is modeled with a DAG of *nonblocking* tasks, analogous to Cilk threads. Such a DAG is illustrated in Fig. 1. Producer cycles are too precious and volatile to waste in a blocked state.

Programming Model

In the *programming model*, the "task server" is the *single* abstraction through which applications communicate with the system. To minimize communication, the application programmer chooses where [de]composition occurs: the consumer, the producer, even the task server, or some combination thereof. For communication efficiency, an application can batch the communication of tasks and computed arguments.

The programmer view is that of a single task server, despite its implementation as a network of servers. The consumer stores a computational task into "the" task server, and receives a callback (`processResult( Object o )`) when the result becomes available. Producers repeatedly take tasks from "the" task server and compute them. See Fig. 2. Such computation results in either the creation of new subtasks and/or arguments that are sent to successor tasks.

The application programming methods for communicating with "the" task server include:

storeTask (Task t): store a task on the task server

storeResult(Task t, int argNo, Object value): store an argument of a successor task on the task server (pseudocode: `t.inputs[argNo] = value`)

The method `processResult ( Object result )` is invoked when a result is available. In the JavaSpace specification, clients cannot compute within the

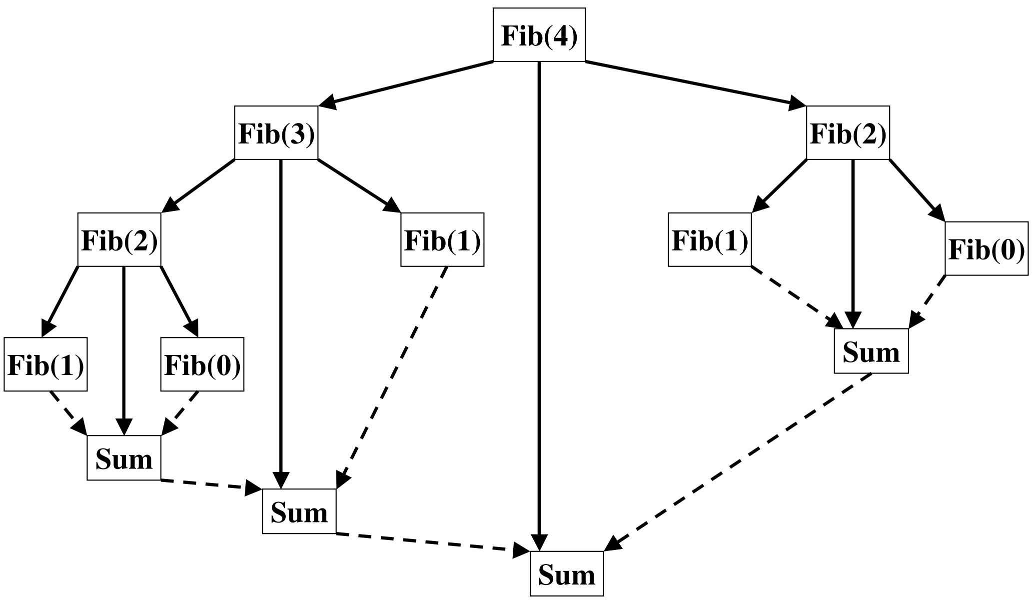

Figure 1: Task DAG for computing the 4th Fibonacci number.

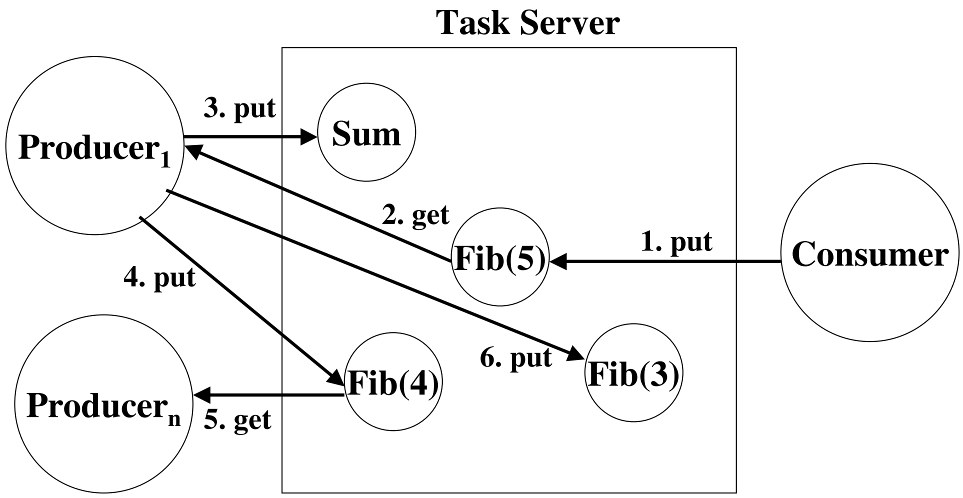

Figure 2: Process communication abstraction. Illustrates the first few tasks of the Fib(5) computation.

space.This is to prevent a client from grabbing the space's computational capacity, which would reduce its responsiveness to other clients. In CX, a production network, a particular set of task servers and their associated producers, executes one computation at a time. Consequently, the application can execute tasks *on* a task server (by setting the Task's boolean executeOnServer member to true). (This is in the spirit of the original tuple space design of the Linda system.). Computed arguments are stored on the server, using `storeResult`. Tasks are *ready* for execution only after receiving all their arguments, if any. For communication efficiency, the above methods have a variant where a *set* of tasks/arguments is stored.

4 Architecture

First, we note some performance constraints. The scheduling mechanisms must be general, subject to the constraint that scheduling operations are of low time complexity: O(1) in the number of tasks and producers. The system must be scalable, high-performance, and tolerate any single component failure. Failure of compute producers must be transparent to the progress of the computation. Recovering from a failed server must require no human intervention and complete in a few seconds. After a server failure, restoring the system's ability to tolerate another server failure requires no human intervention, and completes in less than one minute.

The basic entities relevant to the focus of this paper are:

Consumer (C): a process seeking computing resources.

Producer(P): a process offering or hosting computing resources. It is wrapped in a screen saver or unix daemon, depending on its operating system.

Task Server (S): a process that coordinates task distribution among a set of producers. Servers decouple communication: consumers and producers do not need to know each other or be active at the same time.

Producer Network (N): A robust network of task servers and their associated producers, which negotiates as a single entity with consumers. Networks solve the dynamic discovery problem between active consumers and available producers.

Task servers and production networks are Jini services.

Technological trends imply that network computation must decompose into tasks of sufficient computational complexity to hide communication latency: CX thus is *not* suitable for computations with short-latency feedback loops. Also, we must avoid human operations (e.g., a system requiring a human to restart a crashed server). They are too slow, too expensive, and unreliable.

Why use Java? Because computation time is becoming less expensive and labor is becoming more expensive, it makes sense to use a virtual machine (VM). Each computational "cell" in the global computer speaks the same language.

One might argue that increased complexity associated with generating and distributing binaries for each machine type and OS is an up-front, one-time cost, whereas the increased runtime of a virtual machine is for the entire computation, every time it executes. JITs tend to negate this argument. For some applications, machine- and OS-dependent binaries make sense. The cost derivatives (human vs. computation) suggest that the *percentage* of such applications is declining with time. Of the possible VMs, it also makes sense to leverage the industrial strength *Java* VM and its just-in-time (JIT) compiler technology, which continues to improve. The increase in programmer productivity from Java technology justifies its use. Finally, many programmers *like* to program in Java, a feature that should be elevated to the set of fundamental considerations, given the economics of software development.

There are a few relevant design principles that we adhere to. The first principle concerns scalability: Each system component consumes resources (e.g., bandwidth and memory) at a rate that must be independent of the number of system components, consumers, jobs, and tasks. Any component that violates this principle will become a bottleneck when the number of components gets sufficiently large. Secondly, tasks are pre-fetched in order to hide communication latency. This implies multi-threaded Producers and TaskServers. Finally, we batch objects to be communicated, when possible.

There also is a requirement that is needed to achieve high performance. To focus producers on job *completion*, producer networks must complete their consumer's job before becoming "free agents" again.

The design of the computational part of the system is briefly elaborated in two steps: 1) the isolated *cluster*: a task server with its associated producers, and 2) a producer network (of clusters). The producer network is used to make the design scale and be fault tolerant.

4.1 The isolated cluster

An isolated cluster (See Fig. 3) supports the task graph model of computation, and tolerates producer failure, both node and link. A consumer starts a computation by putting the "root" task of its computation into a task server. When a producer registers with a server, it downloads the server's proxy. The main proxy method repeatedly gets a task, computes it, and, when successfully completed, removes the task from the server. Since the task is not removed from the server until completion notification is given, transactions are unnecessary: A task is reassigned until some producer successfully completes it. When a producer computes a task, it either creates subtasks (or a final result) and puts them into the server, and/or computes arguments needed by successor subtasks. Putting intermediate results into the server forms a checkpoint that occurs as a natural byproduct of the computation's decomposition into subtasks. Application logic thus is cleanly separated from fault tolerance logic. Once the consumer deposits the root task into the server, it can deactivate until it retrieves the final result. Fault tolerance of a task server derives from their replication provided in the network, discussed below.

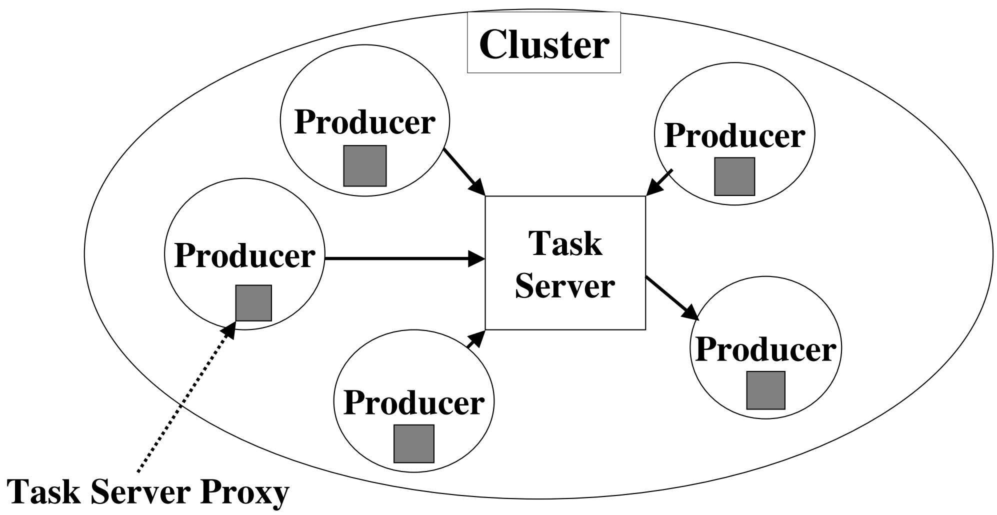

Figure 3: A Task server and its associated set of Producers.

We now discuss task caching. It increases performance by hiding communication latency between producers and their server. Each producer's server proxy has a task cache. Besides caching tasks, proxies copy forward arguments and tasks to the server, which maintains a *ready task heap*: The ordering of ready tasks within the heap is based on 2 components: The dominant component is how many times a task has been assigned. If task A has been assigned fewer times than Task B, then Task A is higher in the heap than Task B. Within that, tasks are ordered by DAG level (see [5]). This minor ordering mechanism is exposed to the application programmer: DAG level is the default implementation of the Task's boolean *isHigherPriority* method. For example, it makes sense to give a Fibonacci task that computes a bigger Fibonacci number a higher priority than a Fibonacci task that computes a smaller number (because the task that computes the smaller number ultimately spawns fewer tasks). In this case, the application programmer can implement the Fibonacci decomposition task's isHigherPriority method accordingly.

When the number of tasks in a proxy's task cache falls below a watermark (see [14]), it *pre-fetches* a copy of a task[s] from the server. For each task, the server maintains the names of the producers whose proxies have a copy of the task. A pre-fetch request returns the task with the *lowest level* (i.e., is earliest in the task DAG) among those that have been assigned the *fewest times.* After the task is complete, the proxy notifies the server which removes the task from its task heap *and from all proxy caches containing it.*

The task server also maintains an unready task collection (of tasks that have not yet received all their input arguments). When a task in this collection receives all its arguments, and hence becomes ready, it is inserted into the ready task heap, and becomes available for pre-fetching. The producer's task cache is organized similarly, with a ready task heap and unready task collection.

Although the task graph can be a DAG, the *spawn graph* is a tree. In Fig. 1, the sub-graph of solid edges is the spawn tree. Hence, there is a unique path from the root task to any subtask. This path is the basis of a unique task identifier. Using this identifier, the server discards duplicate tasks. Duplicate computed arguments also are discarded.

The server, in concert with its proxies, balances the task load among its producers: A task *may* be concurrently assigned to many producers (particularly at the end of a computation, when there are fewer tasks than producers). This reduces completion time, in the presence of aggressive task pre-fetching: Producers should not be idle while other possibly *slower* producers, have tasks in their cache. Via pre-fetching, when producers deplete their task cache, they steal tasks spawned by other producers. Each producer thus is kept supplied with tasks, regardless of differences in producer computation rates. Our design goal: producers experience no communication delay when they request tasks; there always is a cached copy of a task waiting for them (Exception: the producer just completed the *last* task).

4.2 The production network of clusters

The server can service only a bounded number of producers before becoming a bottleneck. Server networks break this bottleneck. Each server (and proxy) retains the functionality of the isolated cluster. Additionally, servers balance the task load ("concentration") among themselves via a diffusion process: Like producers, diffusion of tasks throughout the server network is based on a system of low/high water marks for efficient inter-server communication. Only *ready* tasks move via this diffusion process. Similarly, a task that has been *down*loaded from some task server to one of its producers, no longer moves to other task servers. However, other producers associated with the same task server can download it. This policy facilitates task removal, upon completion. Task diffusion among task servers is a "background" pre-fetch process: Producers are oblivious to it. One design goal: *producers endure no communication delays from their task server* beyond *the basic request/receive latency:* Each server has tasks for its producers, provided the server network has more ready tasks than servers.

We now impose a special topology, that tolerates a sequence of server failures. Servers should have the same mean time between failure as mission-critical commercial web servers. However, even these are not available 100% of the time. We want computation to progress without re-computation in the presence of a sequence of single server failures. To tolerate a server failure, its state (tasks and shared variables) must be recoverable. This information could be recovered from a transaction log (i.e., logging transactions against the object store, for example, using a persistent implementation of JavaSpaces). It also could be recovered if it is replicated on other servers. The first case suffers from a long recovery time, often requiring the human intervention. Since humans are getting *neither* faster *nor* cheaper, we omit human-mediated computer/network administration. The second option can be fully automatic and faster at the cost of increased design complexity.

We enhance the design via replication of task state, by organizing the server network as a *sibling-connected fat tree* (see Fig. 4) We can define such a tree operationally:

- start with a height-balanced tree;
- add another "root";
- add edges between siblings;
- add edges so that each node is adjacent to its parent's siblings.

Each server has a *mirror group:* its siblings in the fat tree. (Since the tree does not need to be complete, it may be that there exists a parent that has only one child. That child uses its parent as its mirror. This is a boundary condition.) Every state change to a server is mirrored: A server's task state is updated if and only if its sibling's task states are identically updated. When the task state update transaction fails:

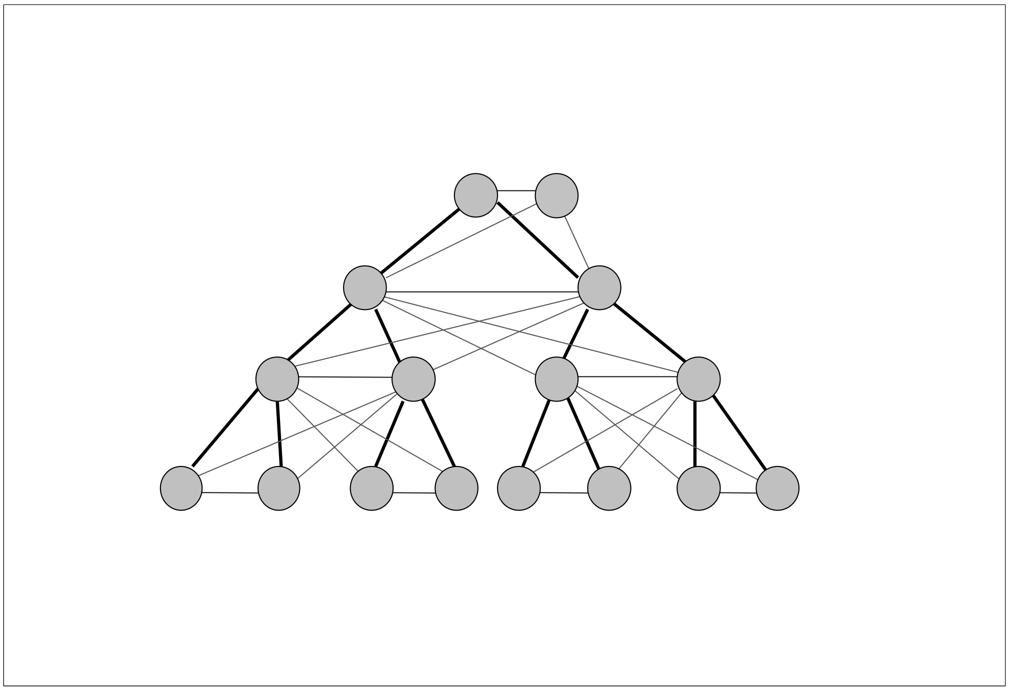

Figure 4: A sibling-connected fat tree.

- The server (or server proxy) that detects the failure notifies the primary root server (the secondary root, if the primary root *is* the failed server).
- Each proxy of the failed server, upon receiving RemoteExceptions, contacts a randomly selected member of the mirror group of the failed server.
- The root directs the most recently added leaf server to migrate (with its associated Producers) to the failed server's position in the network. Its former and new mirror groups are updated to reflect this change.

Automatically reconfiguring the network after a server failure requires $O(B)$ time, where B is the maximum degree of any server, and which is $O(1)$ in the size of the network. When a server joins a network, it becomes the new rightmost leaf in the network. Insertion thus requires $O(B)$ time, independent of the network size.

This design scales in the sense that each server is connected to bounded number of servers, independent of the total number of servers: Port consumption is bounded. The diameter of the network of n servers (the maximum distance between any task and any producer) is $O(\log n)$. Most importantly, the network repairs itself: the above properties hold after the failure of a server. Hence, the network can recover from a *sequence* of such failures.

The consumer submits the "root" task of a computation to the primary root task server. The computation begins when a producer associated with this task server executes this root task, which undoubtedly spawns other tasks. Diffusion then begins.

4.3 Code distribution via the ClassLoader

Omitted from the discussion thus far is our strategy for distributing code. If we make no special provision, task class files are downloaded from the Consumer's codebase. This clearly is a bottleneck, given the degree of parallelism we seek from CX. To scale, the code distribution scheme must have only a bounded number of producers downloading code from any one location, independent of the total number of producers. This implies that the number of download points must increase linearly with the number of producers. There is a natural way to provide for this: Each task server becomes a download location for task class files. The CX class loader downloads task class files to the primary root task server via the consumer's class loader. From there the classes are loaded down through the task server tree. Each producer loads the class files from its task server. This scheme achieves our primary objective: code distribution scales to an arbitrarily large number of producers without a bottleneck emerging.

5 Preliminary experiments

All experiments were run on our Departmental Linux cluster. Each machine has 2 Intel EtherExpress Pro 100 Mb/s Ethernet cards, and is running Red Hat

Linux 6.0 and JDK 1.2.2_RC3. These machines are all connected to a 100 port Lucent P550 Cajun Gigabit Switch.

We tested a CX TaskServer cluster on a doubly recursive computation of the nth Fibonacci number, $F(n)$, augmented with a synthetic workload. Let $T(n)$ denote the number of tasks spawned by computing $F(n)$. Clearly,

$$T(n) = T(n-1) + T(n-2) + 2,$$

with initial conditions $T(0) = T(1) = 1$. By inspecting the dags associated with Fibonacci computation, we see that $T(n) = 3F(n) - 2$. Thus,

$$T(n) = 3\left(\frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{n+1} - \left(\frac{1-\sqrt{5}}{2}\right)^{n+1}\right) - 2.$$

This is the total number of tasks for computing $F(n)$ recursively. The critical path length for $F(n)$ is $2n - 1$.

T_{SEQ} denotes the time for to compute $F(n)$ with a doubly recursive sequential Java program. T_1 denotes the time to compute $F(n)$ with a doubly recursive Java program for CX: each recursive method invocation translates into a task plus the composition task to sum them. Table 1 presents a table of times for workload, T_{SEQ}, T_1, and their ratio, which is referred to as the *efficiency* of the CX application. The times given suggest that, as the workload increases, the efficiency of CX increases. The efficiencies given represent "best" case, since both the producer and its task server were running on the same machine.

Workload	T_{SEQ}	T_1	Efficiency
4522	497420	518816	0.96
3740	415140	436897	0.95
2504	280448	297474	0.94
1576	179664	199423	0.90
914	106024	120807	0.88
468	56160	65767	0.85
198	24750	29553	0.84
58	8120	11386	0.71

Table 1: A table of efficiency (T_{SEQ}/T_1) as a function of the workload for computing $F(8)$. Times are given in milliseconds (ms).

T_P denotes the time for the computation using P Producers. T_∞ denotes the time to complete the computation's critical path of tasks. Thus, as has been reported in the Cilk project:

$$T_P \geq \max\{T_\infty, T_1/P\}$$

To ensure that T_P is dominated by the total work and not the critical path, we thus must have $T_1/P > T_\infty$:

$$P < \frac{3\left(\frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{n+1} - \left(\frac{1-\sqrt{5}}{2}\right)^{n+1}\right) - 2}{2n-1}.$$

For $P = 60$, this inequality holds for $n \geq 14$. Our experiments compute $F(n)$, for $n = [13, 18]$. For larger values of n, total workload would more clearly dominate the time to complete $F(n)$'s critical path.

Traditionally, speedup is measured on a dedicated multiprocessor, where all processors are homogeneous in hardware and software configuration, and varying workloads between processors do not exist. Thus, speedup is well defined as T_1/T_p, where T_1 is the time a program takes on one processor and T_p is the time the same program takes on p processors. The fraction of speedup obtained is the ratio of ideal parallel time over the actual parallel time:

$$\frac{T_1/p}{T_p}$$

We now generalize this formula to a vector of d different processor types, $\mathbf{p} = (p_1, p_2, ..., p_d)$, where there are p_1 processors of type 1, p_2 processors of type 2, etc. The basic idea is simply that $workrate \times time = work$. Let W denote the amount of work to do and R_i denote the work rate for $processor_i$. Clearly, $R_i = W/T_i$, where T_i is the time for the $processor_i$ to complete the work. Ideally, work rates are additive: The work rate for p_1 machines of type 1 plus p_2 machines of type 2 plus ... plus p_t machines of type t is just $p_1R_1 + p_2R_2 + \cdots + p_dR_d$. The ideal parallel time to complete the work thus is

$$\frac{W}{\sum_{i=1}^{t} p_i R_i} = \left(\frac{p_1}{T_1} + \frac{p_2}{T_2} + \cdots + \frac{p_d}{T_d}\right)^{-1}$$

When there is only 1 processor type, the formula above for using p of them reduces to the familiar T_1/p_1. The general formula for the fraction of speedup obtained thus is

$$\frac{\left(\frac{p_1}{T_1} + \frac{p_2}{T_2} + \cdots + \frac{p_d}{T_d}\right)^{-1}}{T_{\mathbf{p}}}$$

While this definition does not incorporate machine and network load factors, it does reflect the heterogeneous nature of the set of machines.

Now, the virtue of having a formula for $T(n)$ comes into play. Clearly, the experiments that take the longest are those that involve only 1 processor, computing $T_1(i)$ for various machines types, M_i. Let $T_1^n(i)$ denote computing $F(n)$ (with an augmented load) on 1 processor of type M_i. We model the computation time of $F(n)$ on a machine of type M_i as a portion of time that is independent of n to start and stop the program, denoted α_i, plus an amount of time that depends on n: $\beta_i T(n)$. That is,

$$T_1^n(i) = \alpha_i + \beta_i T(n).$$

We take actual measurements of $T_1^n(i)$, for 2 values of n chosen such that they result in a system of two independent linear equations. We then solve for α and β. For example, say $T_1^5(i) = 27$ seconds and $T_1^7(i) = 66$ seconds. Then,

$$27 = \alpha_i + 22\beta_i \tag{1}$$
$$66 = \alpha_i + 61\beta_i. \tag{2}$$

Solving, we obtain that $\alpha_i = 5$ seconds and $\beta_i = 1$ second on machine type M_i. We now estimate $T_1^n(i) = 5 + 1T(n)$, for any natural number n. Thus, 2 small experiments minimally suffice for producing a good estimate of a very large sequential execution time. We used this technique to compute the base cases used in the following speedup calculations. This technique obviates the need for extremely large sequential executions that otherwise would be needed to calculate speedups. Large multiprocessor runs require large problem instances. Computing times for the base cases for such runs (e.g., 1000 processor experiments) can, in principle, require many days, even a month of processor time. Thus, using this technique, we avoid the most computationally extended experiments, which are consequently quite precarious (e.g., a momentary power loss requires restarting from the beginning).

Table 2 presents the number of processors of each type that were used in our experiments. Table 3 gives the actual times for 2 synthetic workloads on the

Producers	Dual 512	34
	Dual 1024	22
	Quad	4
TaskServers	Quad	2

Table 2: The number and processors types for Producers and TaskServers.

processor types used in the experiments. We have 3 task types: Decomposition (D), boundary (B), and composition (C).

Dual 512	**D**	**B**	**C**
Workload 1	41	1720	41
Workload 2	41	3650	41
Quad	**D**	**B**	**C**
Workload 1	32	1377	32
Workload 2	32	2925	32

Table 3: Task times, for the 2 processor types. Each had 2 workloads. The 3 task types are decomposition (D), boundary (B), and composition (C). Times are in milliseconds.

The ratio of ideal speedup over actual speedup is less than or equal to 1. Figure 5 shows the ratio of ideal speedup over actual speedup. The figure shows execution times for Fibonacci computations varying from F(13) to F(18). For

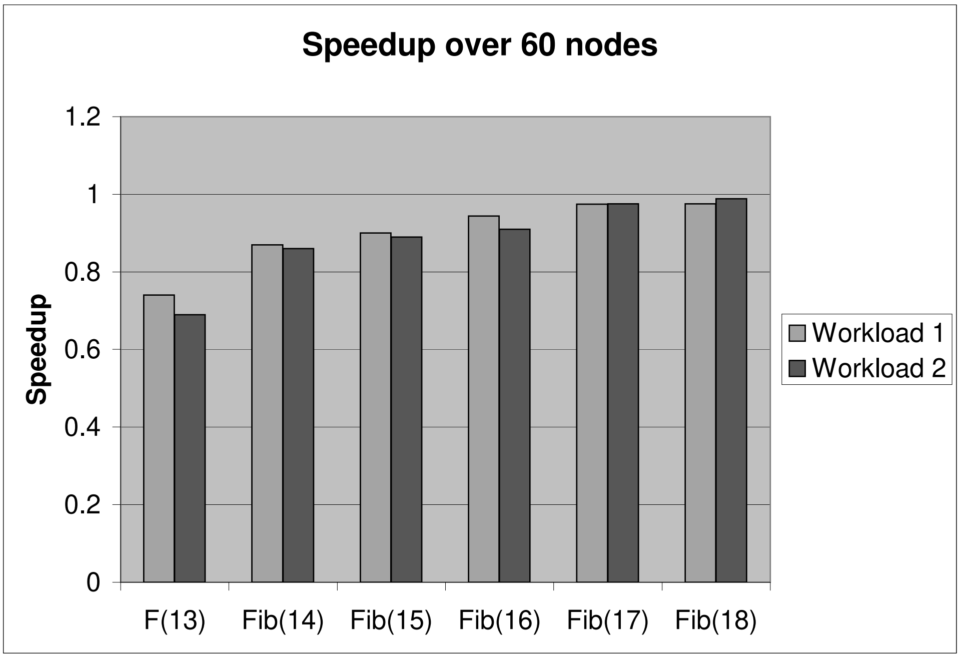

Figure 5: Speedups for computing $F(n), n = 13, 14, 15, 16, 17, 18$ under 2 workloads.

F(13), the ratio of ideal speedup over actual speedup is 0.87. For F(18), the ratio of ideal speedup over actual speedup is 0.99. CX achieves essentially 0.99 of ideal speedup using 60 processors on a complex DAG-structured computation with small tasks (average task time is 1.8 seconds for Workload 1 and 3.7 seconds for Workload 2). This is encouraging: The tasks do not need to be too coarse for respectable speedups. For these preliminary performance experiments, the task servers were not mirroring their state changes. In the final version of this paper, we will report the fraction of speedup achieved with, and without mirroring, to give evidence for the amount of overhead that mirroring incurs.

Figure 6 shows what percentage of idle time was spent during the transient parts of the computation: The initial transient is when the computation begins, and most processors are starving for tasks; the termination transient is when the computation is winding down, and most processors again are starving for tasks. These inevitable transients account for 25% of idle cycles, when the system is achieving 0.99 of optimal speedup. In particular, the idleness due to in initial transient in that case is 0.1% of idle cycles. This suggests that tasks are distributed to the 60 processors with extreme rapidity.

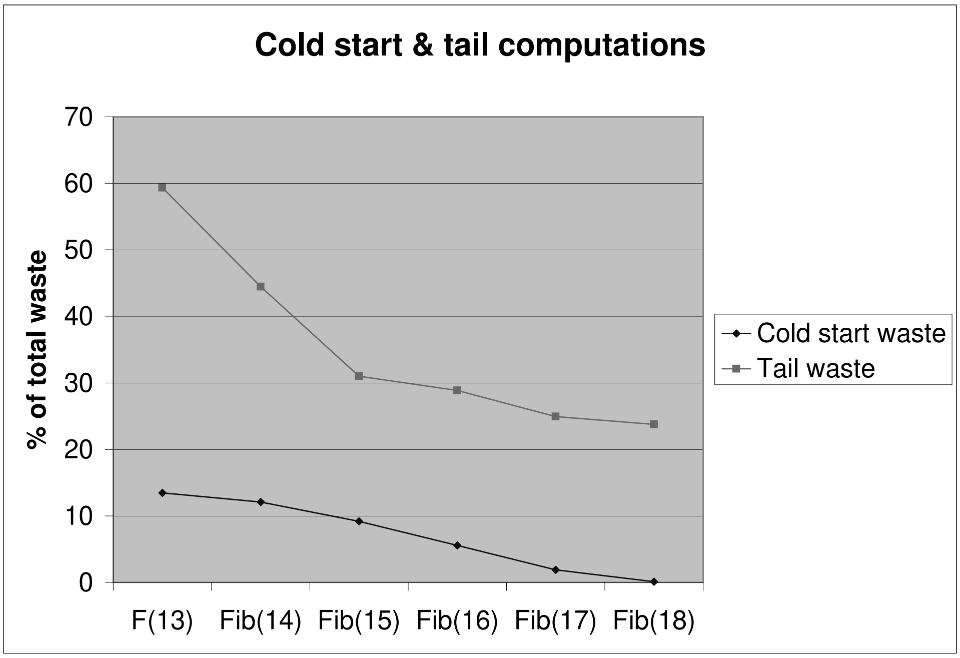

Figure 6: Percentage of idle cycles that are due to start and stop transients, for $F(n), n = 13, 14, 15, 16, 17, 18$ under 2 workloads.

We also performed experiments (on 16 processors) to measure the effect of pre-fetching. For small computations (few tasks and/or short tasks) and fast communication, performance gain via pre-fetching is minimal. As the number of tasks increase and/or the task time increases and/or the communication times increase, pre-fetching helps more and more. Since our cluster has fast communication, we did not obtain data for the case of communications with relatively long latencies. Specifically, for $F(11)$, speedup with pre-fetching was 0.51 of optimal; whereas without pre-fetching, speedup was 0.54. However, for $F(15)$, speedup with pre-fetching was 0.93 of optimal; whereas without pre-fetching, speedup was 0.80. We believe that as the number of tasks increases and/or the task sizes increase and/or communication latencies increase, the benefits of pre-fetching increase commensurately.

6 Conclusion

CX is a network-based computational exchange. It can be used in a variety of environments, from a small laboratory within a single department of a university, to a corporate producer network, to millions of independent producers spontaneously organized into a giant producer network.

We have chosen Java for CX because Java increases application programmer productivity (e.g., is object-oriented, yet serializes objects for communication), reduces application portability and interoperability problems, enables mobile code, will support a high level security API (RMI), and does all this with an acceptable and decreasing penalty vis a vis native machine execution. Jini further reduces CX's administrative complexity to distribute system software component upgrades, and recover from system components whose availability is intermittent.

We believe that our contributions to networked-based, object-oriented parallel computing include:

- The novel combination of variations on ideas by other researchers, including work stealing of non-blocking tasks, eager task scheduling, and space-based coordination.

- A simple, compact API that enables the expression of object-oriented, task-level parallelism. It cleanly separates application logic from the logic that supports interprocess communication and fault tolerance.

- The sibling-connected, fat tree of servers, a recursive, short-diameter, scalable network of task servers that self-repairs in the face of a sequence of faults: The network gracefully degrades from n servers to one server, provided that the failures occur sequentially.

- A simple diffusion process for distributing tasks among the network of task servers. Since in the diameter of the network is $O(\log n)$, the number of edges between any task and any producer is no more than $2 \log n$: Using only local information, task "concentrations" rapidly diffuse into the network.

- The use of task caching/replication and two levels of pre-fetching (including inter-server task diffusion) to hide the large communication latency that is intrinsic to networks.

- A load generator, using the $F(n)$ computation, that strenuously exercises the DAG model of computation: It spawns many tasks that require synchronization of predecessor tasks. This load generator is versatile because it augments the $F(n)$ computation with a parameterized synthetic load. The technique for accurately estimating long sequential execution times, based on 2 short executions, obviates the need for the most time-consuming experiments, saving days of experimental work.

- Providing a test bed for a variety of research topics, such as automated trading, reputation services, authentication services, and bonding services. CX also provides a test bed for algorithm research into network-based parallel computation.

The API can serve as a target for a higher level notation for the object-oriented expression of parallel algorithms. As future work, we are working on an extension to Java, an object-oriented analog to Cilk's extensions to C. The extensions (which, when elided, leave a valid single JVM Java program) could be preprocessed into another Java program—one that exploits the algorithm's task-level parallelism when run on CX's network computing system. We would like to more deeply analyze and experiment with diffusion, modelling task servers and producers as adaptive controllers.

We also would like to experiment with various trading strategies, and program applications for CX that have value to the scientific community.

References

[1] E. Adar and B. A. Huberman. Free Riding on Gnutella. *First Monday*, 5(10), Oct. 2000. http://www.firstmonday.dk/issues/issue5_10/adar.

[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] J. E. Baldeschwieler, R. D. Blumofe, and E. A. Brewer. ATLAS: An Infrastructure for Global Computing. In *Proceedings of the Seventh ACM SIGOPS European Workshop on System Support for Worldwide Applications*, 1996.

[4] 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.

[5] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. In *5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP '95)*, pages 207–216, Santa Barbara, CA, July 1995.

[6] R. D. Blumofe and P. A. Lisiecki. Adaptive and Reliable Parallel Computing on Networks of Workstations. In *Proc. USENIX Ann. Technical Symposium*, Jan. 1997. Anaheim.

[7] P. Cappello, B. Christiansen, M. O. Neary, and K. E. Schauser. Market-Based Massively Parallel Internet Computing. In *Third Working Conf. on Massively Parallel Programming Models*, pages 118–129, Nov. 1997. London.

[8] N. Carriero, D. Gelernter, D. Kaminsky, and J. Westbrook. Adaptive Parallelism with Piranha. Technical Report YALEU/DCS/TR-954, Department of Computer Science, Yale University, New Haven, Connecticut, 1993.

[9] B. O. Christiansen, P. Cappello, M. F. Ionescu, M. O. Neary, K. E. Schauser, and D. Wu. Javelin: Internet-Based Parallel Computing Using Java. *Concurrency: Practice and Experience*, 9(11):1139–1160, Nov. 1997.

[10] E. Drexler and M. Miller. Incentive Engineering for Computational Resource Management. In B. Huberman, editor, *The Ecology of Computation*. Elsevier Science Publishers B. V., North-Holland, 1988.

[11] D. H. J. Epema, M. Livny, R. van Dantzig, X. Evers, and J. Pruyne. A Worldwide Flock of Condors: Load Sharing among Workstation Clusters. *Future Generation Computer Systems*, 12:53–65, 1996.

[12] T. Fink and S. Kindermann. First Steps in Metacomputing with Amica. In *Proceedings of the 8th Euromicro Workshop on Parallel and Distributed Processing*, 1998.

[13] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *International Journal of Supercomputer Applications*, 1997.

[14] E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces Principles, Patterns, and Practice*. Addision-Wesley, 1999.

[15] L. Gong. *Inside Java 2 Platform Security*. Addison-Wesley, 1999.

[16] 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.

[17] K. A. Hawick, H. A. James, A. J. Silis, D. A. Grove, K. E. Kerry, J. A. Mathew, P. D. Coddington, C. J. Patten, J. F. Hercus, and F. A. Vaughan. DISCWorld: An Environment for Service-Based Metacomputing. Technical Report DHPC-042, 1998.

[18] T. Malone, R. E. Fikes, K. R. Grant, and M. T. Howard. Enterprise Computation. In B. Huberman, editor, *The Ecology of Computation*. Elsevier Science Publishers B. V., North-Holland, 1988.

[19] M. O. Neary, S. P. Brydon, P. Kmiec, S. Rollins, and P. Cappello. Javelin++: Scalability Issues in Global Computing. *Concurrency: Practice and Experience, to appear*, 12:727–753, 2001.

[20] M. O. Neary, B. O. Christiansen, P. Cappello, and K. E. Schauser. Javelin: Parallel Computing on the Internet. *Future Generation Computer Systems*, 15(5-6):659–674, Oct. 1999.

[21] R. Scheifler. RMI Security. http://java.sun.com/aboutJava/communityprocess/jsr/jsr_076_rmisecurity.html, August 2000.

[22] 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.

[23] C. A. Waldspurger, T. Hogg, B. A. Huberman, J. O. Kephart, and W. S. Stornetta. Spawn: A Distributed Computational Economy. *IEEE Transactions on Software Engineering*, 18(2), Feb. 1992.

[24] A. Wendelborn and D. Webb. Distributed process networks project: Progress and directions.