Contractor: A Remote Computation Environment for Java™ Applications

Stephen Chenney
http://www.cs.berkeley.edu/~schenney/

Franklin S. Cho
http://www.cs.berkeley.edu/~fscho/

Adam M. Costello
http://www.cs.berkeley.edu/~amc/

16 Dec 1996

1 Introduction

On the internet, and within intranets, there are a large number of under-utilized machines which could be exploited to perform large scale distributed computing. To make these resources available, access must not be restricted to local registered users. However, allowing arbitrary software to run on these machines poses significant security risks and resource management problems.

There are also a growing number of network devices with modest computing capabilities and/or weak network connections. Allowing such devices to offload work to more powerful and better connected remote machines may allow them to perform functions for their users that were previously impossible.

Contractor is a system, built around the Java[1] language and runtime environment, designed to meet the needs outlined above. Its design has been guided by the following goals:

Scalability and Portability: Every machine on the internet should be able to participate.

Safety and Security: Sensitive server resources, such as file systems, must be protected from external tasks. Clients must be able to trust that servers correctly perform the requested tasks.

Balance: Load should be distributed evenly to avoid hot spots, and to ensure that machines are used efficiently.

Fairness: Servers should be fair to both local users and remote tasks. Only resources not required locally should be consumed by external tasks, and free resources should be distributed appropriately among external tasks.

Ease of Use: Access to the infrastructure should be simple, requiring minimal additional effort compared to running a job locally.

Performance: Running tasks remotely must offer a performance advantage over running them locally. The overhead introduced by the infrastructure must not outweigh the added resources provided by the servers.

This paper describes Contractor. It discusses the system architecture, the process of executing jobs, issues of resource management and security, and presents some performance measurements.

2 System Architecture

The complete Contractor system consists of three distinct software modules (figure 1): clients, managers, and workers. Each implements an interface to communicate with other modules, and subsystems to perform specific tasks. Although modules are expected to run on separate machines, this is not a requirement.

[1] Java and Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. The authors of this paper are independent of Sun Microsystems, Inc.

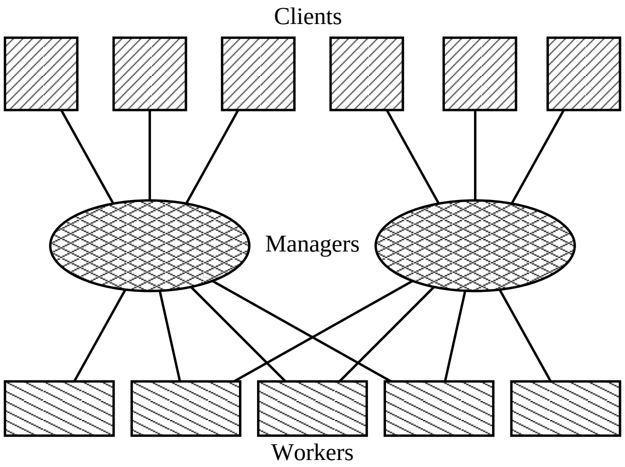

Figure 1: The software modules comprising the Contractor system and the relationships between them.

The separation of the system into modules was designed to achieve the goals of scalability, load balancing, and ease-of-use. Underlying the design were two assumptions about network performance and connectivity.

We assume that client has poor connectivity to the rest of the world. The machines being served are likely to be low powered, and hence unlikely to have good connectivity. Not only may the network connection have poor bandwidth, it may also be intermittent. We would like to avoid communication with the client as much as possible—particularly time-critical communication.

Conversely, we expect the managers and workers to have high connectivity among themselves. High powered machines are likely to have fast network connections, especially if they belong to a cluster. We prefer communication within this group of machines rather than with the client.

2.1 Clients

A client implements an interface for requesting a manager to schedule a remote job. It is the only software module that must be resident on machines that submit jobs. It has three primary responsibilities: processing control messages sent by the manager, responding to requests for Java classes needed by a job on a remote machine, and running a local part of the job that interfaces with the user and local machine. The client's functionality has been kept to the minimum necessary, based on our assumptions of poor performance and connectivity. In effect it manages user I/O and provides data available only from the local machine.

2.2 Managers

A manager implements interfaces for receiving requests for service from clients, and for distributing jobs to clients. In addition, a manager acts as a communication intermediary between workers and clients, and as a caching point for Java classes. We assume that the number of managers will be somewhere between one and two orders of magnitude lower than the number of workers and clients. We also assume that the incoming request load will vary across managers, because managers receive requests from individual clients who have little or no knowledge of the current state of the network.

A client submits a new job by connecting to a well-known port on a manager. The manager guarantees to find a worker to execute the job, or refuse it immediately. When the job is executing, the manager provides a communication interface between clients and workers. It also provides a thread to cache classes requested by the workers executing a particular job.

A worker communicates with a manager through another well-known port. Workers may send a variety of commands to managers:

Announcement informs the manager that the worker is available to execute jobs. It also supplies an integer indicating the maximum number of jobs that the worker will accept from this manager. The manager adds the job to its set of available workers, and thenceforth it may send jobs to the worker. The manager will not deliberately send more jobs to this worker than the maximum number requested.

Adjustment causes the manager to change the maximum number of jobs for this worker. There is no guarantee that the maximum will be honored immediately, because the worker may already be running more jobs (which cannot be revoked) or because of race conditions allowing the manager to send a job while another thread is processing the adjustment.

Submit is a request for service from the worker. It is similar to a client's job submission request, but is used to spawn sub-jobs from within the worker network.

Close informs the manager that this worker would like to stop receiving jobs entirely. The worker should in general set its maximum to 0 before sending this message. A manager does not guarantee that the worker will immediately stop receiving jobs—only that transmission will stop some short time into the future. This is again because of race conditions between threads communicating with the worker.

2.3 Workers

A worker implements the complementary interface to that supplied by managers for workers. A worker announces itself to one or more managers, giving each a maximum allocation of job execution slots. The worker then waits for jobs to be sent from the manager, and guarantees to execute them. The use of multiple managers provides better load balancing properties, because a single worker can adjust its slot allocations to relieve heavily loaded managers dynamically.

The use of a per-manager maximum allows a variety of strategies to be used by a worker in controlling how much work it receives and from whom it is received. One possible extreme strategy is to maintain the invariant that the worker is either running all the jobs it can and has no free slots for any manager, or the free job slots are distributed evenly across managers. This approach ensures close to the optimum number of jobs on the worker, but has a relatively high overhead because maximums are frequently updated. Alternatively, workers can allow themselves to execute more jobs that is optimal by supplying optimistic maximums to managers. This allows a worker to almost always be running as many jobs as possible, but will lead to longer response times for jobs.

The current implementation allocates an equal share of job slots to each manager. The maximums are only changed when one manager uses all the slots is holds, in which case the worker seeks to shift some slots from the least loaded manager to the most loaded manager. This strategy balances the load among managers based on apparent need, with some hysteresis to avoid excess adjustment of maximums.

3 Job Execution

3.1 Remote and Local Jobs

The client software submitting an arbitrary job cannot know in advance all the possible I/O requirements the job may have. For instance, the job may require interactive input, it may need to read and write files, or display image data. Rather than trying to anticipate these requirements, we instead require jobs to be submitted in two parts.

- A **local** part of the job, which will be run on the client machine itself. It is intended that this module form a front end to the main part of the job, supplying appropriate input and processing output. This job may be provided with full access to resources, because it can only affect the local machine. It must implement the Java interface LocalRunnable.

- A **remote** part of the job, which is run by the worker. This module is expected to perform the major part of the computation, receiving its input from the local part of the job, and sending output back. It must implement the Java interface RemoteRunnable.

Each part of the job is supplied with a socket connecting it to the other. The job is free to communicate on the channel in any way, subject to bandwidth limits imposed by the worker.

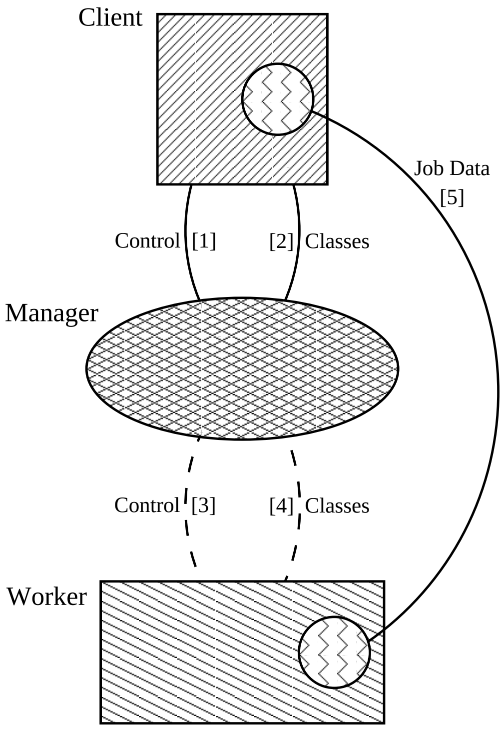

Figure 2: *Connections used in the execution of a single job. Numbers indicate the order of creation. Dashed lines indicate short-lived, on-demand connections.*

The local part of a job must implement a Java interface which defines functions used to communicate with the client thread, and hence the workers running the job. Similarly the remote module must implement an interface for communication with the worker. The interfaces are minimal extensions to the Java interface Runnable.

Converting an existing Java application to a format suitable for execution by the Contractor requires changing the application class to implement the interface RemoteRunnable, and modifying the program I/O to use a socket connection to the local part of the job.

3.2 Running a Single Job

A job is submitted when a client contacts a manager (see figure 2). In return the client receives a job ID by which the job will be identified to all aspects of the system. The connection opened in this initial exchange remains open as the control channel for communication between the manager and client. The initial connection is immediately followed by another which is to be used to download classes. Both the manager and client start threads to manage the requests for classes.

After establishing the initial connections, the manager selects a worker on which to execute the job. Workers are chosen using randomized load balancing—a worker will be chosen randomly such that jobs are distributed according to how many jobs each worker known to that manager is free to run. When a worker is chosen, a control channel is established, and the job ID is forwarded to the worker. Using the control channels, the worker is given the name of the class to run and the port to connect to for servicing of class requests. The client receives the name of the worker that will be executing the job and a port on that worker to connect to.

As the final part of job initialization, the client connects to the worker, thus creating a channel over which job specific data will travel. The worker and client then each start their respective parts of the job, passing on the data socket over which the remote and local parts of the job can communicate.

To actually start the remote part of the job, the worker creates an instance of the top level class for the job using a special class loader. Java semantics are such that this class loader will be used for all classes subsequently required for the job. The class loader we use attempts first to find classes on the worker machine. If the class is not available locally, a request is passed on to the manager, which maintains a thread to cache classes. If the class is found on the manager, it is sent from there, otherwise the request is forwarded to the client. The client loads bytes from its local disk and forwards them to the worker via the manager (which caches the bytes on the way by). This maintains the invariant that each class will be fetched from the client at most once.

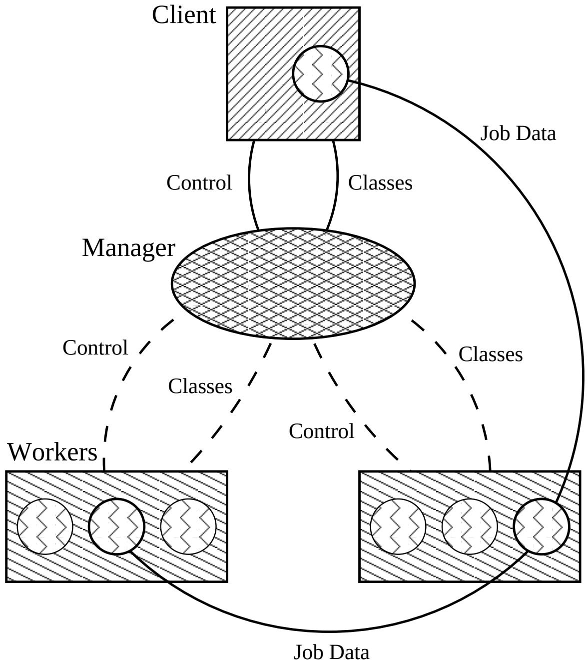

Figure 3: *Connections used in the execution of a sub-job.*

When the job completes, messages are sent along control channels in order to close all connections. Knowledge of job completion is also required for book-keeping, so both the manager and worker can maintain counts of active jobs.

3.3 Spawning Sub-Jobs

Given the parallelism made available by a group of workers, remote jobs must be able to spawn sub-jobs without referring back to the client (whom we are assuming is poorly connected to the network). When a job wishes to spawn a sub-job, it makes a call into the worker running the job. On the job's behalf, the worker submits the sub-job to the manager in control of the spawning job. The process is almost identical to the case of a client submitting a job, with the worker acting as a client.

When a manager receives a sub-job submitted by a worker, it ensures that the the sub-job shares the control thread and class manager of its parent. This has two advantages.

- There is only one control channel to the client for all of the jobs and sub-jobs running on its behalf. This is important because bandwidth to the client is likely to be low.
- All the jobs running on behalf of one client share the same class loader. This has the advantage of a shared cache which significantly reduces traffic to the client.

The data stream for the sub-job is connected to the job that submitted it, as shown in figure 3, again to take advantage of high connectivity within the worker network. From the sub-job's point of view, the relationship between the sub-job and its parent is identical to that between the remote and local parts of a job submitted directly by the client. When a job spawns a child, the data socket is returned, thus preventing the need for special callback mechanisms or the need for jobs to understand any special protocol.

4 Resource Management

Whenever resources are shared, there is a question of how to allocate the resources fairly among competing consumers. This section describes how a single worker allocates CPU cycles, memory, network bandwidth, and disk space among its own jobs. The question of how to allocate resources between a worker and non-Contractor processes running on the same machine is beyond the scope of this paper, and would require features of the underlying operating system.

4.1 CPU Cycles

Each worker contains a CPU scheduler (independent of the virtual machine's thread scheduler). The worker assigns each job a *precedence* (a positive real number), and available CPU time is allotted to jobs in proportion to their precedence numbers.

The worker also assigns each job a requested time quantum, and the scheduler will honor that request within a factor of two. Optionally, the worker can assign each job an upper limit on total CPU time, to cut short infinite loops, deadlocks, or excessive usage.

4.1.1 Implementation

CPU scheduling is provided by the new class ThreadGroupScheduler, which operates on ThreadGroups, an existing Java class for organizing Threads. The worker associates one ThreadGroup with each job, but the scheduler only needs to understand Threads and ThreadGroups, not jobs. A master thread submits ThreadGroups to the scheduler, which suspends all threads in the group. Whenever the master thread calls the scheduler's dispatch() method, the scheduler chooses one group, resumes its threads, sleeps for one quantum (each group may have a different quantum), suspends the threads, and returns. The standard Java Thread and ThreadGroup classes were modified slightly to add an administrative suspend/resume interface (called shackle/unshackle), independent of the existing suspend/resume interface, which can still be used by the application programmer. The dispatcher uses a list of round-robin queues, each traversed half as often as the previous one, to achieve proportional scheduling with average $O(1)$ overhead per context switch. Many more details can be found in the software and its documentation[2].

4.1.2 Precautions

The worker must take a few subtle precautions to make sure that jobs cannot cheat and get more CPU than they are entitled to. It must set the maximum priority of job ThreadGroups to be less than that of administrative threads, so that the dispatch() method can wake up to suspend the threads. The worker should not directly run the RemoteRunnable object loaded from the client, because instantiating that object would give its constructor unlimited CPU time. Instead, the worker should run a wrapper object which instantiates the RemoteRunnable and then invokes the latter's run() method. Finally, the master thread must not make the mistake of believing that the Thread.stop() method actually stops a thread. There is currently no sure way to kill a thread, though the Thread.destroy() method might do the trick in JDK 1.1. Until then, jobs that exceed their CPU time limits must be left suspended.

4.1.3 Limitations

Because we restricted our modifications to those parts of the Java packages written in Java[2], the scheduler suffers from a few limitations. Only one thread group is run at a time, so a multiprocessor may be underutilized by a group with too few threads. Managers could alleviate this problem by choosing compatible workers, if jobs provide hints about how many threads they will run[3]. Even if the scheduler used a much more complex algorithm which kept track of the number of threads in each group and ran multiple jobs at once, it would still be hampered by a limitation of the Java language: There is no way to determine whether a thread is blocked or ready to run. Whenever all threads in the currently running group are blocked, the CPU is wasted. This problem can be alleviated by using smaller quanta, at the expense of increased switching overhead.

4.2 Memory

Java provides no means to discover the size of objects, and no reliable means to be notified when an object is freed (its finalize() method can be called by anyone). Therefore, there is no way to track memory resources without modifying the virtual machine, which we had neither the ability nor the desire to do.

4.3 Network Bandwidth

The worker can assign each job an absolute bandwidth limit (in bits per second) on reads from the

[2] We lack source code for the native methods and the virtual machine, and we want our modifications to be portable.

[3] Service providers could offer economic incentives to users to supply such hints.

network, and a similar limit on writes. Threads that attempt to exceed these limits are forced to sleep just long enough to stay within the limits. Because large reads and writes are more efficient than small ones, the worker may assign each job allowable read and write burst sizes.

4.3.1 Implementation

Network bandwidth limitations are provided by the new class LimitedThreadGroup, a subclass of ThreadGroup. Like the ThreadGroupScheduler, the LimitedThreadGroup knows about Threads and ThreadGroups, but not jobs. Each LimitedThreadGroup maintains two "leaky buckets", one for reading and one for writing, to regulate the rate of data flowing to and from the network. Small modifications were made to the existing classes SocketInputStream and SocketOutputStream, so that on a read or write they ask the LimitedThreadGroup for permission. Permission will always be granted, but not necessarily for the entire number of bytes requested, and possibly after the thread has been put to sleep for a while. Once the actual read or write is performed, the LimitedThreadGroup is told how many bytes were actually transfered, so that unused opportunity can be credited back to the thread group if it hasn't yet leaked out of the bucket. Many more details can be found in the software and its documentation[2].

4.3.2 Limitations

The bandwidth limits are absolute, not proportional, so it is easy to allocate more or less bandwidth than is actually available. However, the limits can be altered on the fly. The CPU scheduler already has problems with blocked threads (see section 4.1), and the bandwidth limitation is achieved by introducing more blocking. No distinction is made between multiple network interfaces, because they are not visible within the Java language.

4.4 Disk Space

Currently, the security manager withholds from every job access to all disk files, making disk space management a non-issue. However, it would be reasonable to allow each job (or each user) access to one directory, and to limit the number and total size of files within the directory. Such a policy could be implemented with additions to LimitedThreadGroup, and modifications to the existing Java classes FileInputStream, FileOutputStream, and RandomAccessFile.

5 Security and Safety

The use of Java as a language and the Java Virtual Machine as a runtime environment provides a large amount of inherent security. For instance, memory access is carefully controlled, and classes are verified as they are defined. To manage other security issues the runtime environment provides two additional hooks.

- The *security manager* is called by many of the basic language classes to verify certain operations, such as file access, the right to instantiate classes, and the ability to define new class loaders. Only one security manager may be defined for any instantiation of the virtual machine.

- The *class loader* loads classes as required by the runtime system. The semantics of the class loader are such that when a class requires another class to be loaded, the same class loader will be used for the new class as was used for the invoking class.

All jobs run by a worker belong to a thread group corresponding to the initial job sent by the client. The security manager uses the thread group to track which threads on the virtual machine are jobs, and limits access to those threads. By preventing jobs from modifying threads outside their own group (and descendent groups) a job cannot subvert the scheduler or the resource limitations, nor can it disrupt the execution of other jobs.

The security manager also refuses to create new class loaders for jobs. This ensures that the same class loader is used for all classes required by the job. The class loader itself, by loading local classes first, refuses to download classes which may subvert the mechanism in some way. File access restrictions complete the picture, by refusing to allow jobs to tamper with local classes.

6 Performance

The key performance metrics for the Contractor are:

- The time to submit and execute a null job, both as an original job from a client, and as a subjob.
- The effect on performance of varying numbers of clients, managers, and workers. It is likely that particular ratios provide optimal efficiency.

The necessary experiments have not yet been performed.

7 Future Work

Contractor has room for enhancement on many fronts.

Client interface A version of the client which is an applet, rather than a standalone application, would be convenient for users, who could then submit jobs to a manager by directing their browser to the manager's web page and typing in the names of their job classes. The greatest challenge is getting around the security restrictions imposed by the browser on the applet.

Fault tolerance There are currently no provisions for gracefully handling network failures or node crashes.

Resource management Perhaps Sun could be persuaded to extend the Thread API so that blocked threads can be identified, allowing more intelligent application-level CPU scheduling. Perhaps they could also be persuaded to extend the garbage collection and/or Object APIs to allow memory usage tracking.

Authentication and accounting Users, managers, and workers should all be mutually authenticated using public key cryptography. In addition to the resource management mechanisms already in place, accounting mechanisms (e.g. access control lists, rate schedules, account balances) are needed to support allocation policies.

Debugging tools Because it is difficult (and sometimes impossible) to get diagnostic messages back to the client from the worker, a debugging tool for remote job classes is needed. It should simulate the execution environment experienced by the remote job. Perhaps an actual worker with an option to allow jobs to access stdout and stderr would suffice.

Hierarchy The client, manager, and worker are implemented separately now, but there is no reason why one process could not provide all three interfaces. Such processes could be organized into a hierarchy. Worker-managers could subcontract jobs out to lower-level workers. Manager-clients, lacking sufficient resources among their own workers, could submit jobs to higher-level managers. Hierarchical organization would greatly improve the scalability of the system.

8 Related Work

There are undoubtedly a large number of related projects on record and in progress. We mention only a few of them here.

ATLAS[1] is a proposal for global network computing with goals almost identical to those of Contractor. It is more ambitious in its checkpointing and migration of jobs. It does not address issues of resource allocation among jobs at a single worker. It is implemented in a combination of C and Java, whereas the Contractor is implemented entirely in Java.

WebOS[6] and Janus[5] (formerly known as SRHA) address some issues of safe execution of untrusted code, particularly authentication and sandboxing. They do not (yet) deal with resource allocation. Janus runs under Solaris, whereas Contractor runs under Java, which is expected soon to be more widely deployed. The WebOS authors are considering using Java in the future.

Penguin[4] is a package for the widely-deployed language Perl that provides authentication and sandboxing services. A system like Contractor could probably be built on Penguin instead of Java, sacrificing some performance and perhaps some resource management in exchange for greater ease of use.

Dynamic Distillation[3] is a technique for improving the quality of network services for poorly connected or low-power clients. A proxy running on a well-connected powerful machine distills content into smaller forms more easily digested by the client. It has been suggested that internet service providers could offer a selection of proxies for use by their modem-bound and wireless customers. If the ISP provided a Contractor, the customers could run an unlimited variety of proxies.

9 Conclusion

The Contractor provides an effective means for users to exploit available computing resources on the internet. The overall system architecture is designed to exploit computing power and connectivity where available while minimizing the use of weak links. It also allows for a broad range of load balancing strategies.

The resource management mechanisms of Contractor allow fair allocation among competing users. The safety features of the Java language and virtual machine are leveraged to protect service providers from inexperienced or malicious users. The result is an efficient, safe, easy-to-use system for opening up the computing resources of the internet.

References

[1] Eric A. Brewer, J. Eric Baldeschwieler, and Robert D. Blumofe. Atlas: An infrastructure for global computing. Obtained from Eric Brewer, 1996.

[2] Stephen Chenney, Franklin S. Cho, and Adam M. Costello. Contractor. http://www.cs.berkeley.com/~amc/research/contractor/.

[3] Armando Fox, Steven D. Gribble, Eric A. Brewer, and Elan Amir. Adapting to network and client variability via on-demand dynamic distillation. We don't know where this was published, 1996.

[4] Felix Sebastian Gallo. Penguin! http://www.eden.com/~fsg/penguin.html.

[5] Ian Goldberg, David Wagner, Randi Thomas, and Eric Brewer. A secure environment for untrusted helper applications. In *Proceedings Sixth USENIX Security Symposium*, July 1996.

[6] Amin Vahdat, Michael Dahlin, and Thomas Anderson. Turning the web into a computer. http://now.cs.berkeley.edu/WebOS/webos.ps, May 1996.