

The Computational Neighbourhood™: A User Centric Approach to High-Performance Cluster Computing

P. Shafae and G. K. Thiruvathukal

DePaul University, JHPC Research Laboratory¹, School of CTI

Abstract

File and printer sharing have been popularized with Microsoft's Windows Network Neighbourhood and the Unix Network File System (NFS). Cluster computing has become an attractive choice for high-performance computing as workstation and network technology continue to become faster and less expensive. Although computer networks can easily be organized into a computing cluster, there is still a need for a simple and easy-to-use programming interface that aids in the construction and deployment of parallel and concurrent applications. The Computational Neighbourhood™ (CN) addresses these needs by pushing past the current limitations of basic file and printer sharing, and allowing users to share the computing power of their desktop. CN takes advantage of Java's heterogeneity and security features to provide a framework that is completely transparent with respect to computing resources. The CN model uses the concept of a job and contention-based scheduling for performing parallel and concurrent computations in a clustered environment. CN also provides a web based interface for submitting, monitoring, and editing parallel and concurrent applications. The current version addresses the issues of user friendliness, while future work will concentrate on security and tuning performance.

Keywords

Desktop supercomputing, sharing, resource management, contention scheduling, relational databases

1 INTRODUCTION

High-performance computing was born out of the need to perform complex and time intensive scientific calculations. To meet this need, computer scientist have for many years attempted to close the gap between computational science and the use of complex computing machinery with more advanced hardware and software. The growing popularity of the networked personal computer has helped tighten this gap. Most of the personal computers that are in wide use by the scientific community run the Windows operating system. Windows's dedication to ease-of-use makes it a popular operating system. Almost anyone can use Windows to create a virtual desktop for organizing electronic documents and data. Windows also allows users to take advantage of disk and printing resources available on a network through the concept of the Network Neighborhood.

Although Windows has made strides in sharing basic disk and printing resources, there still are no simple means for sharing the real power of the personal desktop: its CPU. Many frameworks such as Message Passing Interface[16] (MPI) and Linda have attempted to allow users to utilize the computing power of networked stations. In their attempt to be architecturally neutral, these systems have become complicated to use and program with. In many instances intermediate programming skill are required to use these systems effectively.

¹ Please visit our main web sites at <http://www.jhpc.cs.depaul.edu> for more information on CN and our various activities and projects.

The general scientific community still needs a framework for building and running concurrent and parallel applications with the simplicity and ease-of-use of the Network Neighborhood. Enter the Computational Neighborhood™ (CN). Unlike other high-performance computing frameworks, CN was designed with one main focus: the end user. CN aims to provide an interface similar to the Network Neighborhood for programming a network of workstations[12] (NOW), or commonly termed computational cluster. The decreasing cost of powerful personal computers has allowed many scientific laboratories to own and network workstations. CN provides the means to harness the computational potential of these networked workstations by providing scientist an efficient and user friendly environment for building and performing high-performance computations.

2 PROJECT BACKGROUND

Clusters provide many advantages that directly fit the needs of the scientific community. Personal desktop computers with moderate computing power, network capabilities, and adequate primary and secondary storage units can be purchased for under a thousand dollars. To network these systems, all that is needed is an adequately sized network hub and wiring. Furthermore, a cluster of computers is easily scalable to fit the needs of a workgroup. The addition of a computational node to a cluster is as easy as adding another personal computer.

The management of a cluster is as easy as its setup. Windows and Linux, the two most popular operating systems, provide easy to use interfaces for networking each workstation and managing a private network. Features such as the Network Neighborhood and NFS provide an easy framework for sharing I/O and printing resource. The Computational Neighborhood was designed with the intent to share the computing power of each personal computer with the same ease-of-use.

To reach the design goals for CN, historical programming difficulties such as heterogeneity, security, resource discovery, and process coordination had to be addressed. Heterogeneity has been the center of attention for as long as computers have been networked. Byte ordering and word size are prime example of heterogeneity caused by differences in computer architecture. However, heterogeneity is not limited to architecture design. Varying operating system features, inconsistent compilers, and missing and substituted application libraries are another form of heterogeneity. In conclusion, inconsistencies in both hardware and software result in communication barriers. CN overcomes several aspects of heterogeneity by electing to use the Java programming language which is widely available on many computer architectures and provides a consistent environment for a distributed framework.

Another concern with network programming is security. Networked resources can fall victim in the hands of a seasoned and malicious programmer. Many distributed frameworks rely on cryptography and encryption for protecting the transmission of data. However little can be done to protect a system once a user's name and password have been compromised. The use of Java provides many default security features. In addition, CN uses the concept of Access Control List[1] (ACL) and Capability Lists[1] (CL) to restrict an application's activity. Although the current version of CN only works with Java based applications, future releases will also provide security measures for native applications.

A heterogeneous network consists of computers with varying types and quantities of resources. Locating a certain resource quickly is still a non-standard operation. Resource discovery plays an important part in a framework designed to aid in building parallel and concurrent applications. A programmer typically knows the type and amount of resource that is needed in completing a

computation. The ability to transparently serve resources to users is an important part of creating a user centric and friendly framework. CN utilizes a relational database to track resources available on the network. Users in search of a particular resource will consult the database through a mediator to see if a resource is available. CN's model makes resource discovery transparent to the programmer.

A framework for creating and managing parallel and distributed applications must also provide for coordination between computations. MPI and shared tuple spaces are used to provide parallel applications a means to coordinate with each other. In striving to be general, such frameworks are hard to program. CN, concentrating on ease-of-use, divides the issue of process management into two types: inter-process communication and intra-process communication. Inter-process communication is accomplished with the aid of an application programming interface (API) served by one of the CN components. User's writing CN based application can use the API to await the termination of a previously submitted computation. Intra-process communication is achieved through the use of the Memo system. Memo is a variation on the concept of shared tuple spaces.

3 RELATED WORK

A great deal of work has already been done in the area of parallel and concurrent programming, Much of CN's roots lead to past and present projects.

The concept of a job was introduced by the Uniform Interface to Computing Resource [2] (UNICORE) project. The UNICORE project defined jobs as a set of user operations that are to be performed on a high-performance computing resource such as mainframe. A Network Job Server (NJS) provides an interface by which clients can schedule a job to be performed on a UNICORE resource. CN adopts the concept of the job and a resource repository but concentrates on performing high-performance computations on a clustered environment. With respect to jobs, the Condor[3] project focused on a scheduler that paired jobs to resources based on a "match-making" scheme. The CN framework avoids the complications of load balancing and match-making by allowing resources to compete with one another for jobs that fit their resource profile. However, similar to the Condor project, resource owners may place constraints as to how their resource is to be used. This model ensures that scheduled jobs are not starved and resources are not abused.

The abstract job model is also captured by the Java Beans[4][6] and Enterprise Java Beans[5][7] frameworks. The properties framework of Java is very powerful, allowing for the definition of mutable properties and other elaborations such as indexed properties. While useful, it is not entirely appropriate for modeling properties associated with jobs. A job is not a persistent object and is of a more transient nature. Jobs in the real world tend to be bound to an initial set of parameters, which are read and used to initialize the job code in some way. The job code executes, and a status code is written as an output value. Our approach is that a job has input properties and output properties. The output properties are useful because they can be passed on to another job in a pipeline or graph of jobs.

The Globus[8] project is a framework for sharing very large-scale computational resources and instruments. As pointed out in their tutorial slides and papers, the principle aim is not in clustering. Nonetheless, Globus has many good ideas. The Metacomputing Directory Service (MDS) is a framework for managing the computing resources and dynamic information (such as running jobs). MDS utilizes LDAP[9][10] for organizing resource profiles. There are some disadvantages of using LDAP vs. a relational database, especially for any dynamic aspects, since

LDAP has been designed primarily to address a high volume of reads and a very low (infrequent) volume of writes. Globus also provides interfaces to off-the-shelf job schedulers (the GRAM and RSL components); however, in most clustered environments, access is not scheduled and really lends itself to a different model than provided by a centralized batch scheduler (which has its origins in mainframe computing). The CN framework addresses resource discovery via a contention-based scheduling paradigm where resources contend for work. This has an advantage of giving us an approximation to load balancing (which in its pure formulation is an NP-hard problem).

When it comes to coordination, there are basically a number of ways to go:

- Linda-like tuple spaces (Linda, JavaSpaces, JavaNow – one of our previous works)
- MPI
- Parallel Virtual Machine (PVM)

All three of these approaches are excellent vehicles for coordination but suffer from the disadvantage of requiring a great deal of user knowledge of the underlying system, despite the occasional mention of the word “virtual”. Globus has addressed (successfully) many issues associated with starting jobs on nodes and a number of advanced issues, such as co-scheduling; however, the interfaces are very low-level in many regards (most of the system has been written in C). CN allows jobs to be defined at a very high-level, using a graphical interface to a Java API, wherein the *nature* of the job is specified; parallel, process farm, etc. When defining a job, the user can specify what kind of job it is and a process called binding (similar to the idea employed in language implementation) instantiates the components to enable that kind of job to run best. CN uses a message passing system called Memo[14][15] in its first generation, which supports the framework of Linda plus active semantics (active tuples) with dataflow semantics. It is beyond the scope of this paper to detail every aspect of the Memo API; however, it has been designed to support latency-tolerant coordination, since latency is likely to remain a reality in practical clustered environments.

Jini and JavaSpaces (a component of Jini found in the `jini.space` package) provide a framework for defining a services architecture. Key features of Jini include resource “discovery” and “lookup”. Interfaces are stored in the Jini space to provide meta-data about devices and their capabilities. JavaSpaces itself is used to provide a coordination framework for distributed device usage. Key features include the core coordination primitives found in Linda with a significant deletion: `eval()` is missing. In place of `eval`, however, is `notify()` which provides a form of distributed event processing. Subscribers can process the events asynchronously. Memo provided a similar feature with its `putDelayed()` method, which is basically a message forwarding mechanism. CN is a services architecture that is geared toward high-performance computing. It has many similar components with a lightweight design. RMI has been replaced with GMI (a simplified RPC framework from RMI aimed at peer-to-peer vs. pure client/server), implemented transactions (JavaSpaces makes no guarantees about transactions--we do), a much improved coordination model over Linda, called Memo, which supports active tuples and asynchronous operations.

Databases have long been shunned by the high-performance computing community. The current state of the art is to support LDAP (as done in Globus) or an alternative information service. Relational database technology has been adopted in our approach. We consider it a very attractive option, especially in the management of user job data. Databases provide two key features: persistence and fault-tolerance[19]. Relational databases are easily replicated and use file objects to track information. If a system failure occurs, data stored in the database can be used to recover the state of jobs or automatically restart jobs. Current databases and distributed

databases support advanced features, such as replication, logging, and transactions. These features are essential for building a robust framework that scales. Technologies such as LDAP, although powerful for directory representation, have a long way to go before such robustness is achieved. As clustering is increasingly becoming of industry interest (as well as academe), it will be an expectation that industry standard technologies are used for high-end computational environments.

3.1 Remote Method Calling

CORBA, RMI, and DCOM [1][20] are all standards for distributed object computing. Numerous papers have been published in the Grande series and web site about performance inefficiencies associated with serialization and ways to overcome them. Generic Method Invocation (GMI) is a component of CN that attempts to overcome the problems of serialization. A key difference of GMI from RMI is its focus on peer-to-peer rather than pure client/server. This allows all peers to communicate as equals, as in CORBA, which does a good job in this regard. CORBA was not intended for an environment in which most of the communication is Java-to-Java, for much associated baggage is carried for addressing aspects of mapping to other languages such as C and C++. DCOM, a Microsoft-centric framework, will not work well for our purposes, since the cluster is assumed to run Windows and some flavors of Unix machines, mostly Solaris and Linux. DCOM (being based on the DCE/RPC framework) does have the advantages of being much like RPC, which is extremely light compared to CORBA and RMI.

The approach with GMI is to provide the light encoding of RPC, the peer-to-peer model of CORBA, and the Java-to-Java nature of RMI. The current prototype has been able to achieve this; however, future plans are to expand GMI into a new distributed object system that takes the best of all worlds and leaves behind the rest.

4 OVERVIEW OF THE CN FRAMEWORK

CN is a highly modular framework with three main components: jobs, the DeviceClient, and the JobAgent. The following will discuss the roles of each component, their relationship, and their physical design.

4.1 Jobs

In the CN framework, a job is defined as a unit of work that a user wants to perform. Specifically, it is the user's application wrapped to fit the specifications of a CN job and packaged into a single JAR file.

CN distinguishes between three type of jobs. The simplest type of job is one that requires a single processor. Although high-performance application rarely require only a single processor, it is commonly the case that a single processor job needs to run before all others to setup a shared space or populate an already initialized shared space.

The next type of job is one that requires more than one processor, but no coordination is necessary among the sub jobs. An example of such a job would be an application that calculates the dot product of two vectors. Several jobs can be started where each job works on a non-overlapping segment of the vectors. Each sub result can then be combined to obtain the dot product of the two vectors.

Finally, users may specify a multiprocessor job where coordination is needed among the sub jobs. An example is the Laplace heat equation. The parallel version of this application requires that each job, with an assigned part of the problem, coordinate with other sub jobs after each relaxation iteration.

4.2 What is a CN Job?

The CN framework is designed to simplify the adaptation of existing Java applications into a CN job. As an example, assume we have the following “Hello World” application as specified in Figure 1. This application is also an example of a single processor job.

Figure 1: HelloWorld Application

```
public class HelloWorld {
    public HelloWorld() {
        System.out.println( ``HelloWorld object constructed!`` );
    }
    public static void main( String [] args ) {
        HelloWorld hw = new HelloWorld();
        System.exit( 0 );
    }
}
```

When executed, this application will display a string that reads “HelloWorld object constructed!”. To adapt this application to run in the CN framework a new method needs to be defined that has the following signature:

```
public static Properties <method-name> ( Properties in_props );
```

This method will essentially perform the same operation as `public static void main(. . .);` it will serve as the thread of execution. For the revised HelloWorld application see Figure 2.

Figure 2: HelloWorld Application Adapted for CN

```
import java.util.Properties;

public class HelloWorld {
    public HelloWorld() {
        System.out.println( ``HelloWorld object constructed!`` );
    }
    public static void main( String [] args ) {
        HelloWorld hw = new HelloWorld();
        System.exit( 0 );
    }
    public static Properties myMainStatic( Properties in_prop ) {
        HelloWorld hw = new HelloWorld();
        return new Properties();
    }
}
```

The new method, named `myMainStatic`, fits the specified CN signature and performs the same operation as the `main` method. The revised HelloWorld job is now ready to be executed within the CN framework by submitting the compiled source code to CN via the CN web interface. The name of the class, `HelloWorld`, and the name of the CN signature method, `myMainStatic`, is all that is needed to load the application inside the framework.

4.3 Jobs in Detail

The HelloWorld example illustrates how simple it is to convert existing applications to work within the CN environment. Inspection of the required method's signature reveals that a Properties object is a required parameter. Properties, an extension of Hashtables, store a set of key/value pairs. The value of a key can be obtained by using the `getProperty()` method, which takes a String parameter as the key and returns a value of type String.

At execution time, the CN environment passes a set of default key/value pairs to the user's job via the Properties object (in the HelloWorld example, this object is named `in_prop`). These default key/value pairs can be used by the user's code to obtain information on the current job. Figure 3 lists a subset of key/value pairs passed by default.

Figure 3: Subset of Default Key/Value Pairs

```
user.id=[user who submitted the job]
user.job.id=[universal job id: a unique ID assigned to every job]
user.job.proc_num=[virtual processor to identify a sub job]
user.job.total_procs=[total number of processors in a job]
user.job.classpath=[the classpath used when launching the job]
user.job.main_method=[method that fits the CN signature]
user.job.main_class=[class containing method with CN signature]
```

As an example, the following operation is used to obtain the unique job id associated with the HelloWorld application during run-time:

```
String myJobId = in_prop.getProperty( ``user.job.id`` );
```

Besides the default key/value pairs, the user may define a custom set of key/value pairs at the time of job submission. The CN web interface allows users to define any custom variables in the following format:

```
EXAMPLE_KEY=EXAMPLE_VALUE
EXAMPLE_KEY2=EXAMPLE_VALUE2
```

These key/value pairs are then passed to the application along with the default set. Properties allow the user to pass any number of arguments to the submitted application in a manner similar to the way that command line arguments are passed to the `static void main` via an array of Strings.

Properties objects are also used to store any values returned by the user's application. A user defined job can create a new instance of the Properties object and use it throughout the application to store various return values

The next example application, see Figure 4, will display how a multiprocessor job is defined and how a job uses a Properties object to return values. The example application performs an integration of a line concurrently by breaking the computation into sub regions. Each region of the computation will be worked on by a separate processor, but there will not be any coordination between the regions. The user can then take the results of each region and obtain the final value for the integration.

Figure 4: Trapezoidal Integration

```
import java.util.Properties;

public class IntegTrap {
    private float start_x;
    private float end_x;
    private int granularity;
```

```

Function f;

/** Constructor */
public IntegTrap( float start_x, float end_x, int granularity, Function f ) {}

/**
 * Integrate the function f between the indicated starting and
 * ending interval by calling the f() method of the function.
 * The granularity value is used to determine the number of
 * trapezoids that will be used to estimate the area under the
 * curve.
 * @return The area under the curve of a specified region
 */
public float integrate() {}

public Properties myMainStatic( Properties in_prop ) {
    Properties out_prop;
    Function func = new myFunction( );
    int result;

    if( (in_prop.getProperty( ``user.job.proc_num`` )).equals( ``0`` ) ) {
        IntegTrap it = new IntegTrap( 0.0, 5.0, 50, func );
        result = it.integrate();
        out_prop = new Properties( new ByteArrayInputStream(
            new String( ``RESULT_0.0_to_5.0=`` + result ) ) );
        return out_prop;
    }
    else if( (in_prop.getProperty( ``user.job.proc_num`` )).equals( ``1`` ) )
        IntegTrap it = new IntegTrap( 5.0, 10.0, 50, func );
        result = it.integrate();
        out_prop = new Properties( new ByteArrayInputStream(
            new String( ``RESULT_5.0_to_10.0=`` + result ) ) );
        return out_prop;
    }
    else {
        out_prop = new Properties();
        return out_prop;
    }
}

public class myFunction implements Function{
    public float f( float x ) {
        return 2*x + 5;
    }
}

public interface Function{
    public float f( float x );
}
}

```

The IntegTrap application requires two processors to integrate the area under a line. From the user's standpoint, there is one physical application that performs the entire computation. However, the virtual processor id assigned to the job is used to integrate a only specific region of the function. This model allows for applications to be easily parallelized.

This example also illustrates the use of a Properties object for reporting the final result. In each of the two else clauses, the user creates a new Properties object, chooses an arbitrary key, and reports the result of the integration. When the entire job runs to completion, the user can inspect the returned key/value pairs to obtain the final solution for the integration.

5 The Job Life Cycle

Each CN job follows a general life cycle. A CN job comes to existence when a user defines the method with the appropriate signature and packages his code in a JAR file. In the current version, the user must utilize the web interface (Figure 5) to submit the job to a particular cluster. The first two form fields require the user to supply a username and password. Each cluster provides limited access to a set of trusted users. The implementation of this account concept will be explained in more detail with the discussion of the JobAgent.

Figure 5: Job Submission HTML Form



Next, the user must publish the name of the method that meets the CN job signature and the name of the class containing this method. These two pieces of information will be used to dynamically load and execute the user's code. Optionally the user can assign a name and a brief description to the job. These fields are not used in executing the code, but prove useful for organizing multiple submissions. The user must then provide the code that has been prepared for execution. The web interface allows the user two methods of locating the code: a URL or a local file system path. Finally, the user is asked to specify any additional parameters and the number of processors that are needed to perform the computation. The parameters can simply be the command line arguments needed to run the application.

It is important to note that the current version of CN does not allow the use of any non-standard Java packages. In the event that package dependencies do exist, the user can include the package with their source code as one JAR file. Future versions will allow users to specify any package dependencies as well as the location of these packages.

Once the job is submitted, the information obtained from the user will be passed to a CGI script that is authored in the Python scripting language. The script uses the information in the HTML form to schedule a CN job. Scheduling is performed by passing the user information to the CN JobAgent. The JobAgent is the component responsible for managing user and job information. The JobAgent is accessed by the JobAgent API. The design and implementation details of the JobAgent will be described at a later section. At this point it is important to note the role of the JobAgent within the job life cycle.

Once the script has scheduled the job with the JobAgent, the job's state is declared as "submitted". The user can use the web interface to check the status of all scheduled jobs. Each available computational node in the cluster running the CN framework is executing the DeviceClient component. This application is responsible for obtaining a submitted job, executing the job, and then reporting the exit status and return values of the job.

Once a job is retrieved by the DeviceClient, several different outcomes may ensue. The user's code may terminate successfully with various output properties. On the other hand, the user's code may encounter errors and exit improperly. In either case, the DeviceClient uses the job's output properties to reports the exit status and return values. The job's status is changed to indicate either "done_ok" or "done_failed" based on the exit status of the code. All errors that occur with the job are logged inside the DeviceClient log file. The completed job rests in the CN framework until the user decides to remove the code and job information via the web interface.

Figure 6: Job Life Cycle



Multiprocessor jobs follow the same life cycle. However, once a multi-part job is scheduled, it is not declared "done" until all sub parts have been completed. The specific mechanism is as follows. The JobAgent uses the scheduling information to track the total number of processors a particular job requested. Every time that a DeviceClient requests a job, the JobAgent will first return all of the scheduled parts of a multiprocessor job. In addition, every time that a

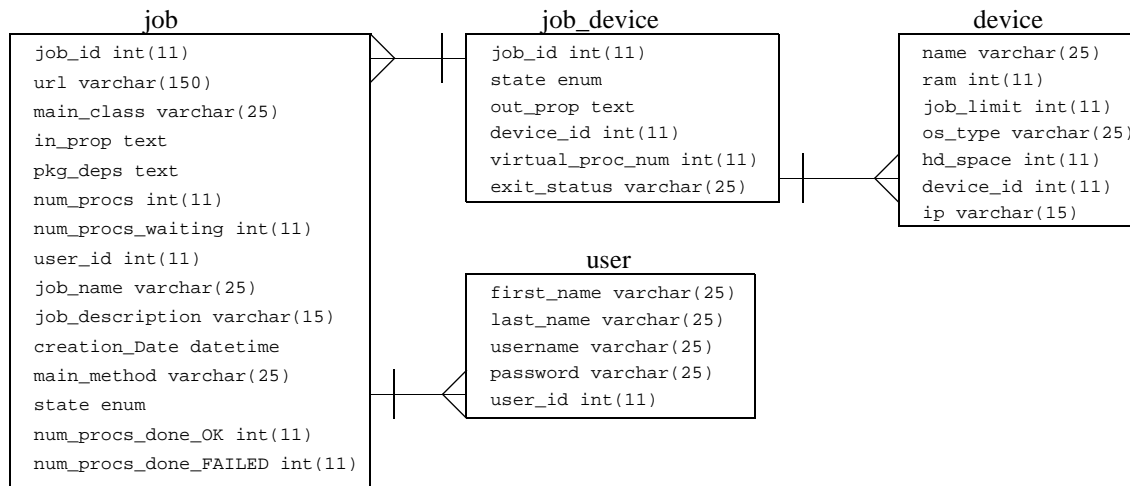
DeviceClient reports the completion of a job, the JobAgent checks to see if all sub-parts are completed and the exit status of each part.

5.1 JobAgent Design and Implementation

The JobAgent is the component responsible for managing user and job information inside a relational database. The relational database used is the freely distributed MySQL[11] database. An important aspect of using a relational database is the notion of a virtual user. No user of the CN framework has an actual system account, instead information is kept in a table. During job submission, the username and password are authenticated via the JobAgent against the database record. Threats against system security are minimized since no system accounts are used.

Three other tables are kept in the database (see Figure 7). The job table is used to record the initial information on a job. The table has fields to track all of the information that the user provides during job submission. The job_device table is used to track a job once it has been scheduled to be worked on by a DeviceClient. When a device client requests a job from the JobAgent, the number of processors required in the job table is reduced by one, and a new entry is made in the job_device table. This is done until the total number of processors that a job needs reaches zero, indicating that all sub parts of a multiprocessor job have been scheduled to be worked on by DeviceClients. The device table is used to track the properties of DeviceClients that are registered with the system. This table is also used to track which DeviceClients are working on which jobs.

Figure 7: JobAgent Database Tables



By virtue of design the JobAgent ensures that all access to the database are performed in an atomic fashion, even though MySQL does not support the notion of transactions. This allows a single JobAgent to serve multiple web interfaces and DeviceClient requests without corrupting the database records.

The JobAgent also provides an API that is made available by a socket based messaging interface, the General Method Invocation (GMI). A discussion of the design of GMI is not the focus of this paper. The JobAgent creates a GMI based RemoteCallServer to handle API connections. Any Java client, such as the DeviceClient, that would like to use the API calls the getConnection() method provided by the static class JobAgentAPI. This method returns an instance of a RemoteJobAgent object that provides the various JobAgent API methods.

The client can then call the methods locally and the `RemoteJobAgent` object performs the remote call. In this situation GMI provides a very light and simple method of performing remote method invocation, without the complication of stubs, skeletons, and method versions.

Figure 8: Subset of the JobAgent API

```
void connect( String host, int port, String user, String password);
void disconnect();
int createJob( String job_name, String job_description, String source_url, Properties
in_prop, int life_time );
boolean abortJob( int job_id, int virtual_proc );
String checkJobStatus( int job_id, int virtual_proc );
void waitForJob( int job_id, int virtual_proc );
void indicateJobCompletion( Properties out_prop );
int getDeviceID( String host, String host_name );
```

The API allows clients to schedule a job, obtain job information, and await the completion of a job. These basic methods lead to the next most important aspect of the API: the ability to perform inter-job coordination. Any job can potentially use the JobAgent API to schedule other jobs and wait for their completion. Similar to the Unix operating system, the CN framework provides the ability to fork and wait for child jobs. This is an important feature since in many job data-flow paradigms there is a need to create new jobs and wait for their termination.

5.2 DeviceClient Design and Implementation

The DeviceClient is responsible for requesting and executing jobs. One of the most important design features of the DeviceClient is the model by which jobs are obtained. Jobs are received and worked on based on a contention scheme; DeviceClients compete with one another for jobs.

DeviceClients can be configured to control the work load and network activity of the cluster node. A configuration file defines properties such as the location of the JobAgent, the number of jobs allowed in the job queue, and the sleep interval between job requests. These properties allow the owner of the node (computational resource) to limit the extent to which the resource is used.

When a job is obtained by the DeviceClient, a new `Job` object is created to manage its execution. The `Job` object is responsible for creating a temporary work directory for the user code, obtain the user code from the specified URL or file location, and fork a new instance of the Java Virtual Machine (JVM) to process the user code. A new JVM instance is created by using the Runtime class `exec` method to execute the `JobRunner` application. The `JobRunner` application invokes the method that fits the CN method signature using reflection. When the user's code runs to completion, the `JobRunner` passes the output `Properties` to the appropriate `Job` object. The `Job` object then notifies the DeviceClient regarding the termination of the job, which is then reported to the JobAgent.

The DeviceClient provides a means by which a resource owner can monitor and manage jobs that are currently in the job queue. A resource owner would use DeviceClient's `ps` command to inspect the queue and its `kill` command to terminate a running job.

6 Status and Future Work

All of the source code for the current version of CN is available through our CVS repository located at: <http://www.jhpc.cs.depaul.edu/cgi-bin/cvsweb.cgi>. There is little information available for installing and running the application, although more clear directions and supporting documentation will be provided by March of 2000.

The next version of CN will include a distributed JobAgent model. This model will allow JobAgents to communicate with one another to resolve the location of a particular job. Much like the DNS model, when the job is found, the JobAgent that is responsible for managing that job will be cached and called again when querying that job. The distribution of the JobAgent allows joining multiple clusters and running multiple instances of the JobAgent on a single cluster to distribute the load.

The security model of CN will also be implemented. The current design has allowed room for the implementation of Access Control Lists (ACL) and Capability Lists (CL). These features are adopted from the Multics and Hydra operating systems, respectively. Java currently uses the `SecurityManager` class to authenticate all operations that an application attempts to perform with regard to I/O. The `DeviceClient` will be adapted to use the ACL information on a user to overload the `SecurityManager` class and limit the user's I/O capabilities. Similarly, Capability Lists will be used to define the access right to various shared resource. ACL and CL will provide a strict governing of shared resources.

User defined packages will also be implemented. At the time of submission, users will be able to select any custom packages that their code depends on, or upload a package that is not listed. This feature allows users the flexibility of working with any package needed to perform a desired calculation.

7 Acknowledgments

We wish to acknowledge the reviewers...

Bibliography

- 1 A. S. Tanenbaum, A. S. Woodhull. Operating Systems: Design and Implementation, Prentice Hall, NJ, 1997, pp. 448-450.
- 2 UNICORE - managed and maintained by Genias HmbH and Pallas GmbH at <http://www.fz-juelich.de/unicore>
- 3 Condor - managed and maintained at <http://www.cs.wisc.edu/condor/>
- 4 D. Flanagan. Java In a Nutshell, O'Reilly, Sebastopol, CA, 1997, pp. 178-199
- 5 D. Flanagan, J. Farley, W. Crawford, K. Magnusson. Java Enterprise In a Nutshell, O'Reilly, Sebastopol, CA, 1999, pp. 15-78, 174-222
- 6 Sun Microsystems, Inc., Java Beans - on line information <http://www.javasoft.com/beans>
- 7 Sun Microsystems, Inc., Java Enterprise Beans - on-line information at <http://www.javasoft.com/j2ee/white.html>
- 8 Globus - developed and maintained by MCS at Argonne National Laboratory at <http://www.globus.org/>
- 9 "Linda Introduction", part of the LINDA TUTORIAL, Scientific Computing Associates on-line, 1996-1997
- 10 T. Howes, M. Smith. LDAP: Programming Directory-Enabled Applications with Lightweight Directory Access Protocol, Macmillan, Indianapolis, IN, 1997
- 11 MySQL on-line documentation and source maintained at <http://www.mysql.com/>
- 12 T. E. Anderson, D. E. Culler, and D. A. Patterson, "A case for NOW", *IEEE Micro*, February 1995.
- 13 Z. Chen, K. Maly, P. Mehrotra, R. K. Vangala, and M. Zubair, "Web Based Framework for Distributed Computing," *Proceedings of ACM Workshop on Java for Science and Engineering Computation*, Las Vegas, NV, June 1997

- 14 W. T. O'Connell, G. K. Thiruvathukal, and T. W. Christopher, "Distributed Memo: A Heterogeneously Parallel and Distributed Programming Environment," *Proceedings of the 23rd International Conference on Parallel Processing*, August 1994
- 15 W. T. O'Connell, G. K. Thiruvathukal and T. W. Christopher, "The Memo Programming Language," *Proceedings of the International Conference on Parallel and Distributed Computing Systems*, October 1994
- 16 S. W. Otto, M. Snir, and D. Walker, "An Introduction to the MPI Standard" In J. Dongarra, CS-95-274, January 1995
- 17 Sun Microsystems, Inc., JavaSpaces Specification, <http://java.sun.com/products/javaspaces/>
- 18 Sun Microsystems, Inc., JINI Specification, <http://www.sun.com/jini/>
- 19 M. T. Ozsü and P. Vlduriez, *Principles of Distributed Database Systems*, Second Edition, Prentice Hall, Upper Saddle River, New Jersey, 1999
- 20 D. L. Galli, *Distributed Operating Systems: Concepts and Practice*, Prentice Hall, Upper Saddle River, New Jersey, 1999