Ninflet:
a Migratable Parallel Objects Framework using Java

Hiromitsu Takagi
Nagoya Institute of Technology
takagi@center.nitech.ac.jp

Satoshi Matsuoka
Tokyo Institute of Technology
matsu@is.titech.ac.jp

Hidemoto Nakada
Electrotechnical Laboratory
nakada@etl.go.jp

Satoshi Sekiguchi
Electrotechnical Laboratory
sekiguti@etl.go.jp

Mitsuhisa Satoh
Real World Computing Partnership
msato@trc.rwcp.or.jp

Umpei Nagashima
Ochanomizu University
umpei@is.ocha.ac.jp

Abstract

Ninflet is a Java-based global computing system that builds on our experiences with the *Ninf* system which facilitated RPC-based computing of numerical tasks in a wide-area network. The goal of Ninflet is to become a new generation of concurrent object-oriented system which harness abundant idle computing powers, and also seamlessly integrate global as well as local network parallel computing. Ninflet is designed to make use of Java features to implement important features in global computing, such as resource allocation, inter-Ninflet communication, security, checkpointing, object migration, and easy server management via HTTP.

1 Introduction

World-wide computer networks have made it possible to delegate tasks to the computers on opposite sides of the globe, allowing us to make use of powers of idle computers. Furthermore, PCs and workstations are becoming increasingly employed in a clustered manner, sometimes with gigabit-capable local-area networks (LAN). The clusters could be some dedicated parallel computer, or just a set of nodes in an office environment[1]. Either way, much of their computational resources would often be idle (say, during non-office hours), and an computing environment which makes use of such idle clusters seamlessly would be highly desirable.

We are conducting an ongoing research project on global computing called *Ninf (Network Infrastructure for global computing)*. [1] [2] [3] Ninf is a client-server system makes available multiple remote computing and database servers on the network, allowing clients to semi-transparently call remote numerical applications and libraries from languages such as Fortran, C, and Java, and even DCOM objects. We have already made publically available the first version of the Ninf system (`http://ninf.etl.go.jp/`), and conducted extensive benchmarking on real networks[4], and also have built adaptors[5] to similar global computing systems such as Netsolve[6].

Ninf is intended to reuse existing libraries and applications under heterogeneous, multi-lingual setting in a coarse-grained manner. On the other hand, it is not very well suited when a client wants to program his application with finer-grained objects in a more orderly manner as is with parallel object-oriented languages such as HPC++ and MPC++. For example, Ninf only supports RPC-style invocation of large, coarse-grained libraries, and have no support for objects of medium-sized or smaller granularity. The *metaserver*, one of whose task being a resource manager, does not take into account placing tightly-communicating parallel tasks on a cluster, as it has no knowledge of communication patterns between the spawned libraries. Also, Ninf currently lacks security features as well as task migration, which make it difficult to place and appropriately control computing servers server on every node in a cluster. Although the latter feature could be implemented by taking a approach similar to Condor[7], the migratory site has to be homogeneous for binary compatibility.

In order to provide a more uniform, finer-grained object-based programming model, we decided to construct *Ninflet*, overcoming the limitations of Ninf. In its design, we have established the following criteria:

- The system must be easy install and manage across a number of cluster nodes, and furthermore must autonomously accept incoming computing requests using arbitrary programs. For a dedicated computing servers hosting a fixed set of libraries as is with Ninf, it would be relatively effortless for administrators to install the required libraries and manage the server. However, in global computing setting where arbitrary code could be uploaded and executed on a cluster, it would cause substantial administrative overhead to install the libraries, and manage hundreds of independent nodes in a cluster.

- The client programming model should be flexible

[1]Hereafter, we use the term “cluster” to indicate both dedicated and non-dedicated locally-networked set of PCs and workstations.

enough so that it would accommodate a variety of programming styles with good abstractions, while being based on a portable, widely-available software platform. Designing a customized (object-oriented) programming language for is a conceivable approach, but goes against the intention of widespread acceptance and usage. Rather, we must exploit how much we can leverage the existing language platform, in our case being standard Java, to provide multiple programming models to the client user without language extensions or customized runtime.

- The system must be secure, especially against malicious uploaded client code, while being flexible enough to allow efficient parallel computing between nodes in the cluster as well as distributing part of tasks to other portions of the global network. More specifically, it must be secure enough such that files on the server node cannot be arbitrarily modified or deleted, but at the same time, must be flexible enough so that computation can be distributed across security firewall.

- The system must prevent global computing tasks from intruding upon foreground user jobs of the server node, almost to the point where it should be basically unnoticeable that the node is being actually provided as a server for global computing. Thus, the system must, for example, have provisions for scheduling the period of time when the computing servers are made available for private use or public use with different policies, and furthermore, migrating the global computing task immediately when the owner starts unscheduled usage of the node.

- The system must provide good support for instable and dynamically changing nature of the underlying computing environment. Thus, the client user must be able to cope with distributed computing issues such as fault tolerance, load balancing, remote error checking and recoverly, etc.

Early on in the design process, we have decided to employ Java without language extensions or VM modifications. Java already facilitates the following characteristics for distributed computing:

- The `SecurityManager` class controls local file access, network connection, access to threads of other ThreadGroups. Furthermore, `SecurityManager` is extensible, allowing the system builder to establish his own security models.

- The `ClassLoader` class provides separate namespace to classes with identical class names.

- The language is pointer-safe and array access is bounds-checked.

- The bytecode verifier checks 'illegal' code, such as those with type errors introduced after compilation by direct modification of the class file.

Although Java Applets make use of the above features to implement safe execution of downloaded code, we claim that they are too low-level and furthermore too restrictive as a global computing platform. Facilities for RPC in Java such as RMI or HORB[8] also are too primitive themselves to realize the above design goals, although they can be used as a platform for building a more robust infrastructure.

We have designed and implemented our Ninflet prototype to meet the above design criteria:

- The Ninflet system is easy to install, manage and update. The entire package can be downloaded and installed by a novice user. For cluster environments, each server in the Ninflet system has HTTP management capabilities through a standard browser, and furthermore allows the user to manage multiple computing servers collectively from a single server site.

- The programming model is such that a set of uploadable and distributable concurrent objects (called '*Ninflets*') are defined using standard Java facilities. Ninflets are automatically distributed to appropriate hosts by a *Dispatcher*, taking into account network topologies and server availability/load. It utilizes the low-level Java features such as RMI, without extending the language or the JVM.

- The Ninflet system establishes its own security model by extending the `SecurityManager` to protect itself from malicious uploaded client code, while allowing direct inter-Ninflet communication as well as indirect across-firewall communication between the clusters.

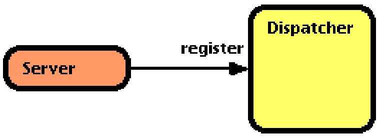
- In order to support Ninflet evacuation as well as object migration and fault tolerance, the Ninflet system facilitates checkpointing mechanism for Ninflets. Also, the system can schedule the availability of computing servers, and forces Ninflets to be either evacuated, migrated, or terminated if foreground user tasks mandate full usage of the node. In the event of a computing server failure, Ninflets can be re-started from their checkpointed state.

The rest of the paper proceeds as follows: Section 2 overviews the Ninflet system, Section 3 gives an example of Ninflet programming using a simple Master-Worker pattern, Section 4 covers the management capabilities of the servers, Section 5 discusses related work, and we conclude in Section 6.

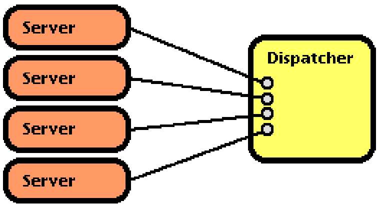
2 Architecture of the Ninflet System

2.1 Ninflet Startup and Client Program Execution

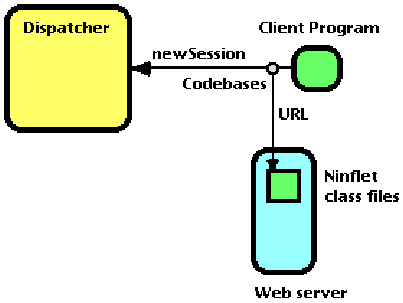
The Ninflet system consists of three components, the *Ninflet Server*, the *Ninflet Dispatcher*, and the *Client*



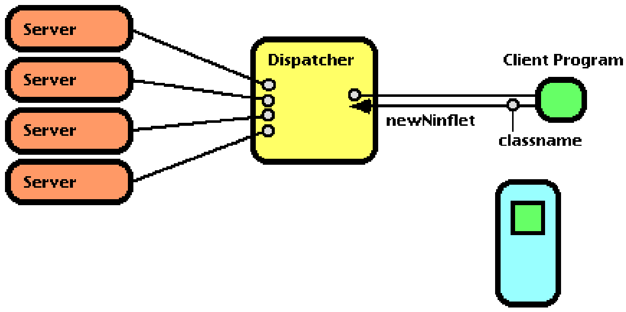
(a) A Server registering itself to the Dispatcher.



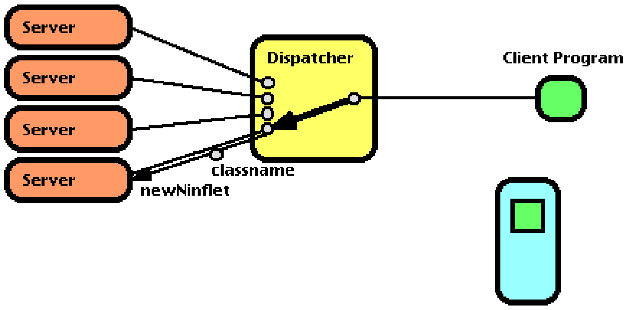
(b) A set of Servers are registered to the Dispatcher.



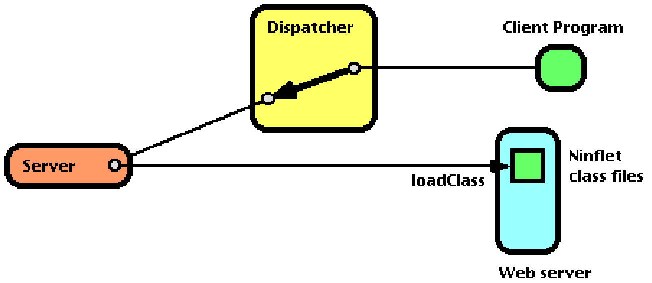
(c) A Client connecting to the Dispatcher



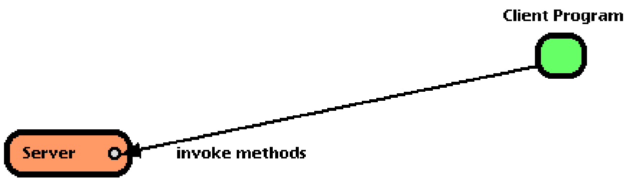
(d) The Client launching the Ninflet.



(e) The Dispatcher selecting an available Server.



(f) The Server loading and instanciating the classes.



(g) The Client calling the Ninflet directly.

Figure 1: The Launching Process of Ninflet

Program. In addition, standard Web components, such as HTTP/FTP servers and Web browsers, are utilized.

The 'user' of the Ninflet system could play two types of roles, one being the *resource provider* (or *server provider*), and the other being the *client programmer*. The resource provider will have the Ninflet Server daemon running on his host. On launching the Server, the provider specifies a Ninflet Dispatcher daemon running on some host by specifying its URL. Thereafter, the given Server will be managed by the specified Dispatcher(fig.1(b)), and the Ninflets to be executed will be uploaded from the Dispather to the Server (fig.1(a)). Dispatcher daemons will be launched by the administrator of the Ninflet system, typically who manages the cluster in LAN, or a set of clusters in WAN.

The Client programmer designs his program by largely dividing it into two components: one being the Client program, which will be executed on the user's host and acts as a master control program; the other being the Ninflets, which describe the main portions of the program to be executed in parallel, and actually are programmed as concurrent objects executed on the Servers. The Ninflets are defined as subclasses of `JP.go.etl.ninflet.Ninflet`, and compiled to create their respective classfiles. Then, the Ninflet classes are placed on a downloadable directories of the Web or FTP server. The Client program is written such that the references to the Ninflet classes for instantiation are made through the URL(fig.1(c)).

When a client user wants to execute his parallel program he initially invokes his Client program on his host machine, specifying the URL of a given Dispatcher of his acquaintance. The Client program, in turn, will obtain a (remote) RMI reference to the Dispatcher, and requests the creation of a `Session` instance, giving the `Codebase` instance as an argument. Here, the *Codebase* instance is a table of pairs consisting of (1) package name of Ninflet classes, and (2) the corresponding URLs of the node where the Ninflet classfiles are located. The *Codebase* instance is used by the Server to find out the location of Ninflet classes to load.

Figure 2 shows an example Client Program. Notice the initialization of the `Codebase` object which indicate that classes in the package `JP.ac.nitech.center.takagi.ninflet` are located at the `Codebase` URL `http://www.center.nitech.ac.jp/~takagi/ninflet/`. This codebase object is being passed to the Dispatcher as an argument of the `newSession` method.

Then, the Client program requests the Dispatcher to instantiate a new Ninflet on one of the Servers selected according to some criterion (typically, some node in the same LAN cluster as the Dispatcher) by calling the `newNinflet` method of the Dispatcher. The Dispatcher, in turn, selects a Server from the available Servers pool according to the criteria, and requests the creation of an instance of the Ninflet class as specified by the Client program (fig.1(e)).

The Server then creates an instance of the `NinfletClassLoader` class, which acts as a customized loader of the specified Ninflet class. Class loading is done via a socket connection using the Codebase object info to locate the Ninflet classfile (fig.1(f)). Other required runtime classes are loaded on-demand by the custom class loader.

Finally, the new instance of the loaded Ninflet class is created, and its remote reference is returned to the Dispatcher and the Client. After its initialization, the Client can call the Ninflet directly.

2.2 Administration of the Server

The Ninflet server can either accept or deny the request form the Dispatcher to dispatch a Ninflet for execution. This setting can be changed even while both the Server daemon and Ninflets are operating, using a Web browser interface — the Ninflet server itself facilitates a simple HTTP server capability which allows various controls of its behavior using a Web browser.

The most basic setting is instantaneous change of Ninflet acceptance/denial. The provider of a Ninflet computing server can allow the acceptance of Ninflets at his discretion. When in denial mode, the Dispatcher does not request creation of new Ninflet instances to the server. Also, it is possible to change the settings automatically using timers to, say, allow the use by the server only during the night or on weekends.

If a Ninflet happens to be executing when the server is changed to denial mode, an *evacuation* message is sent to the Ninflet. The action of the Ninflet upon reception of the message will be covered in the next section; here we mention that, such evacuation action may take some time, resulting in a Ninflet unwantedly consuming computational resources. To overcome this to some degree, the provider of the server may set a *stay timer*; When the Ninflet to be evacuated is still consuming computational resources even after the stay timer has expired, the Thread of the Ninflet will be forced to terminate.

```
::::::::::::::
Main.java
::::::::::::::
package JP.ac.nitech.center.takagi.ninflet.examples.matmult;
import JP.go.etl.ninflet.client.Client;
import JP.go.etl.ninflet.Codebase;
import JP.go.etl.ninflet.dispatcher.Dispatcher;
import JP.go.etl.ninflet.dispatcher.Session;
import JP.go.etl.ninflet.dispatcher.LANClusterResourceManager;
import java.net.*;
public class Main {
  public static void main(String[] args) {
    Dispatcher dispatcher = Client.getDispatcher(
      "rmi://ninf.etl.go.jp/ninflet/dispatcher"
    );
    Codebase codebase = new Codebase();
    codebase.add(
      "JP.ac.nitech.center.takagi.ninflet",
      "http://www.center.nitech.ac.jp/~takagi/ninflet/"
    );
    Session session = dispatcher.newSession(
      codebase, new LANClusterResourceManager()
    );
    Matrix m1;
    Matrix m2;
    int numOfWorkers;
//...(abbreviated)
    NinfletStub master = session.newNinflet(
      MatMultMaster.class, new MatMultMasterArg(
        numOfWokers, m1, m2
      )
    );
    NinfletResult r = master.getResult();
    Matrix m3 = ((MatMultMasterResult)r).product
//...(abbreviated)
  }
}
```

Figure 2: An Example of Client Programming.

We have designed the Ninflet system so that, using these features, it would not be disadvantageous for the provider of a Ninflet server even if long-running Ninflets are accepted. We feel such a provision is necessary so as to overcome the skepticism on the provider's side that his computing abilities will be substantially penalized.

2.3 Checkpointing and Ninflet Migration

As mentioned above, the system must accommodate sudden termination of Ninflet execution in order not to penalize the server providers. On the other hand, especially in scientific codes, computation tends to be long-running and thus there would be a great possibility that Ninflet will be terminated, resulting in substantial loss for the client.

In order to cope with such termination, we provide a checkpointing mechanism for each Ninflet, which implements Ninflet evacuation from the server. Evacuation occurs when there is an evacuation message from the Server to the Ninflet, the state of the Ninflet being saved to the location indicated by the Dispatchers a result. The evacuated Ninflet could resume execution on a different server later on. Also, the Dispatcher could also order the Ninflet to be migrated immediately so as to continue its execution; in this case, the state of the object will not be stored into stable storage, but rather, will be copied to a new Ninflet instance on a different Server.

Checkpointing is a scheme often used in long-running scientific computing. Although automatic checkpointing using compiler support has been proposed[9], the current version of Ninflet requires explicit checkpointing by the Ninflet programmer. More specifically, the programmer inserts `checkpoint()` method calls at an appropriate place in a computational loop. With this, the superclass `NinfletImpl` writes the object into stable storage. It is the user's responsibility to skip over already computed operations when a Ninflet execution resumes from a snapshot state.

There are three levels of checkpointing, according to the granularity of actually taking the snapshot:

Level 1
: Preserve the object state every time `checkpoint()` is called.

Level 2
: Clone the content of the current state once when the `checkpoint()` is called, and copy the cloned state into stable storage when the evacuation message is sent.

Level 3
: Basically do nothing at `checkpoint()`, and when the evacuation start message is received, the object state is written into stable storage when the next `checkpoint()` is called.

These different checkpointing schemes could be used in the following way, depending on the nature of the algorithms: Level 1 Ninflets would be resilient to server crashes and network failures, but incurs substantial overhead. Level 3 Ninflets would incur least overhead, but the computation could be terminated if the stay timer were set to short duration by the Ninflet server providers. The use of Level 2 would allow good compromise, but might incur some wasted computation.

In order to support transparent resumption of execution of evacuated object on a different host, the object returned by newNinflet constructor is actually an additional stub object which wraps the RMI stub. This allows automatic updating of remote references that points to the evacuated/resumed Ninflet. When the Dispatcher notifies the other Ninflets and clients that some Ninflet has migrated, the host specified in the wrapper stub acts as a placeholder for updating the remote reference. We also use the mechanism for the case when the Server no longer responds and results in RemoteException. In such a case, the Dispatcher is ordered to recreate a Ninflet on a different server and restart the execution from its checkpoint; again, in such a case, the stub is updated to point to the new server.

2.4 Ninflet Security Model

As mentioned earlier, Ninf places strong emphasis on the importance of security models for safe remote execution, so that the server providers would be less reluctant to provide services due to security concerns. For this purpose, the Ninflet system imposes a customized security model based on existing Java security features.

More specifically, the Ninflet security model is defined by `JP.go.etl.ninflet.server.NinfletSecurityManager`, and embodies the following security policies:

- Prohibit network connections other than RMI.
- Prohibit creation of windows by AWT.
- Prohibit manipulation of threads.

Also, as is with `AppletSecurityManager`, we prohibit access to the local filesystem, `System.exit()` and creation of a new `ClassLoader`.

One of the major differences between Ninflets and Applets is that, We allow RMI connection to hosts other than the one where we loaded the Ninflet classfile from. This is more lenient than the Applet security model, where connections are only allowed to the host of the Applet codebase. On the other hand, we limit connections to instances to subclasses of Ninflet. Otherwise, it would allow arbitrary intrusion of other RMI applications within the firewall.

3 Applying Ninflet for Parallel/Distributed Computing

3.1 Parallel/Distributed Execution of the Master-Worker Pattern

The *Master-Worker* pattern is typically applied when a problem can be subdivided into small pieces. The Worker is programmed to handle the subdivided pieces, and the Master divides the given problem into pieces, creates Workers, dispatches the problem appropriately, and then receives and combines the results from the Workers, obtaining the final result.

Here, we implement a parallel matrix multiplication using the Master-Worker pattern of Ninflet as showing in Figure 3. We have prepared the `Master` and `Worker` classes as a class library in the package `JP.go.etl.ninflet.lib.masterworker`. They are shown as figure 4. The `Master` class has four *Template Methods*[10] which will be overridden by its subclasses. Both the `Master` and the `Worker` classes are subclassed to define the specific behavior of the problem.

We define the main loop of the matrix multiplication in the `run()` method of the subclassed `MatMultWorker`, and the `init()` method initializes the sub-problem according to the parameters. The `MatMultMaster`, which is subclassed from the `Master` class, specifies the Worker class with the `getWorkerClass()` method, and describes the subdivision of the main matrix multiplication by the `getWorkerArg()` method, and finally combines the results of submatrix computations from the worker using the `combineResultOf()` method.

MatMultMaster as well as MatMultWorker instances are executed on Ninflet servers as Ninflets. MatMultMaster is launched by the Client as was shown in Figure 2.

3.2 Experiments and Results

As a preliminary evaluation, we executed MatMult Ninflet on the ETL-Wiz cluster. ETL-Wiz is a dedicated cluster of 32 Alpha workstations each running at 333Mhz, and interconnected with a 100Base/TX Ethernet switched network. We used DIGITAL's JDK V1.1.5 with JIT compiler and native threads as the platform JVM.

Figure 5 shows the relative speedup in parallel execution of the MatMult Ninflet. The horizontal axis indicates the number of workers which are executed in parallel. Each line indicates the various size of the n by n matrix n is varied from 200 to 1000. We can observe that, When n becomes small, execution time of each Worker becomes negligible and overhead of Ninflet creation becomes dominant. When the number of workers becomes large, data transfer overhead becomes significant. The result shows that Ninflet instantiation time is still significant for smaller-grained problems where Ninflet is supposed to be differentiated from the Ninf system, and a more efficient creation and dispatch mechanisms need to be worked on, while not sacrificing security issues.

4 Service Management

The Ninflet system can be used not only to construct a global computing environment, but also could be used for implementing a virtual parallel computer on a cluster of workstations and PCs on a LAN. In this case, the Ninflet Dispatcher would be running on one host, and each of the other hosts would be running the Ninflet server. We would restrict access to the Dispatcher, so that it would not be accessible from outside the LAN.

The Ninflet system has an additional feature to reduce the management of the Servers in this case. The Dispatcher embodies server management and monitoring features, allowing easy aggregate control of the Servers. A user can connect to the Dispatcher via a Web browser, displaying various server status as well as changing their settings.

A Dispatcher can also connect to other Dispatchers, allowing formation of arbitrary topologies of a network Dispatchers, such as trees or rings. This allows load distribution amongst multiple Dispatchers, avoiding the possible bottlenecks when the client requests concentrates on a single Dispatcher. Dispatchers mutually communicate various information regarding the servers

```
::::::::::::::
MatMultWorker.java
::::::::::::::
package JP.ac.nitech.center.takagi.ninflet.examples.matmult;
import JP.go.etl.ninflet.Ninflet;
import JP.go.etl.ninflet.NinfletArg;
import JP.go.etl.ninflet.lib.masterworker.Worker;
import java.rmi.RemoteException;
public class MatMultWorker extends Worker implements Ninflet {
  public MatMultWorker() throws RemoteException {}
  Matrix m1, m2, m3;
  protected void init(NinfletArg arg) {
    super.init(arg);
    m1 = ((MatMultWorkerArg)arg).m1;
    m2 = ((MatMultWorkerArg)arg).m2;
    m3 = new Matrix(m1.rows, m2.cols);
  }
  public void run() {
    for (int r = 0; r < m1.rows; r++) {
      for (int c = 0; c < m2.cols; c++) {
        double sum = 0;
        for (int k = 0; k < m1.cols; k++) {
          sum += m1.value[r][k] * m2.value[k][c];
        }
        m3.value[r][c] = sum;
      }
    }
    result = new MatMultWorkerResult(m3);
  }
}
::::::::::::::
MatMultMaster.java
::::::::::::::
package JP.ac.nitech.center.takagi.ninflet.examples.matmult;
import JP.go.etl.ninflet.Ninflet;
import JP.go.etl.ninflet.NinfletArg;
import JP.go.etl.ninflet.NinfletResult;
import JP.go.etl.ninflet.lib.masterworker.Master;
import JP.go.etl.ninflet.lib.masterworker.Worker;
import java.rmi.RemoteException;
public class MatMultMaster extends Master implements Ninflet {
  public MatMultMaster() throws RemoteException {}
  Matrix m1, m2, m3;
  float subSize;
  protected void init(NinfletArg arg) {
    super.init(arg);
    m1 = ((MatMultMasterArg)arg).m1;
    m2 = ((MatMultMasterArg)arg).m2;
    m3 = new Matrix(m1.rows, m2.cols);
    subSize = (float)m1.rows / (float)this.numOfWorkers;
  }
  protected Class getWorkerClass() {
    return MatMultWorker.class;
  }
  protected NinfletArg getWorkerArg(int workerNum) {
    int row = (int)(subSize * (float)workerNum);
    int nextRow = (int)(subSize * (float)(workerNum + 1));
    int rowLength = nextRow - row;
    Matrix m1Sub = m1.getSubmatrix(row, rowLength, 0, m1.cols);
    return new MatMultWorkerArg(m1Sub, m2);
  }
  protected void combineResultOf(
    NinfletResult r, int workerNum
  ) {
    int row = (int)(subSize * (float)workerNum);
    int nextRow = (int)(subSize * (float)(workerNum + 1));
    int rowLength = nextRow - row;
    m3.putSubmatrix(
      row, rowLength, 0, m3.cols,
      ((MatMultWorkerResult)r).product
    );
  }
  protected NinfletResult createMasterResult() {
    return new MatMultMasterResult(m3);
  }
}
```

Figure 3: Matrix Multiplication Using the Master-Worker Pattern

```
::::::::::::::
Master.java
::::::::::::::
package JP.go.etl.ninflet.lib.masterworker;
import JP.go.etl.ninflet.Ninflet;
import JP.go.etl.ninflet.NinfletImpl;
import JP.go.etl.ninflet.NinfletStub;
import JP.go.etl.ninflet.NinfletArg;
import JP.go.etl.ninflet.NinfletResult;
import java.rmi.RemoteException;
public abstract class Master extends NinfletImpl {
  protected int numOfWorkers;
  public Master() throws RemoteException {}
  protected void init(NinfletArg arg) {
    numOfWorkers = ((MasterArg)arg).numOfWorkers;
  }
  public void run() {
    NinfletStub[] workers = new NinfletStub[numOfWorkers];
    Class workerClass = getWorkerClass();
    for (int i = 0; i < workers.length; i++) {
      NinfletArg arg = getWorkerArg(i);
      workers[i] = this.session.newNinflet(workerClass, arg);
    }
    for (int i = 0; i < workers.length; i++) {
      combineResultOf(workers[i].getResult(), i);
    }
    result = createMasterResult();
  }
  protected abstract Class getWorkerClass();
  protected abstract NinfletArg getWorkerArg(int workerNum);
  protected abstract void combineResultOf(
    NinfletResult result, int workerNum
  );
  protected abstract NinfletResult createMasterResult();
}
::::::::::::::
Worker.java
::::::::::::::
package JP.go.etl.ninflet.lib.masterworker;
import JP.go.etl.ninflet.NinfletImpl;
import java.rmi.RemoteException;
public abstract class Worker extends NinfletImpl {
    public Worker() throws RemoteException {}
}
```

Figure 4: The Implementation of the Master-Worker Library.

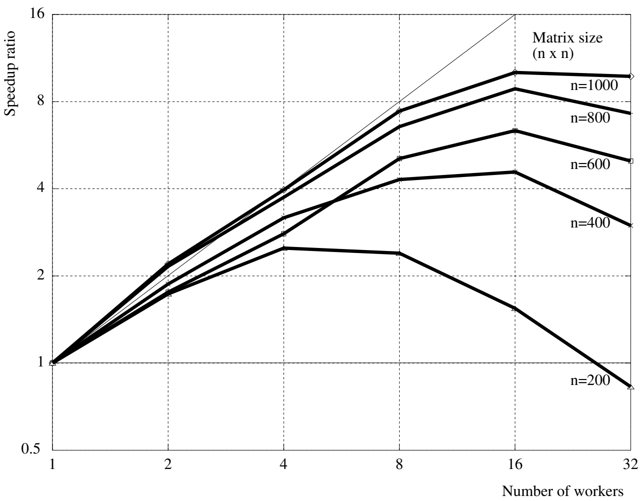


Figure 5: Speedup Ratio of MatMult Execution.

they are managing, such as machine types, availabilities, loads, mutual network bandwidths, etc.

For example, one could create a set of Dispatchers, each of which manages its own workstation- or PC-cluster. These can be connected to a master dispatcher which represents an organization such as a University or a company[2] The master Dispatcher communicates with each other and exchanges information regarding the availability of the parallel computational resources within the organization. For instance, the usage of clusters within an organization could be restricted to intranets during the day, whereas they could be made available for outside usage during the night.

5 Related Work

There have been several work which have made collaborative use of computational resources over a global network. For example, in the RSA Data Security Secret-Key Challenge [11], the distributed.net project[12] solved a 56bit RC5 in about 200 days using approximately 15,000 to 30,000 Windows PCs and Macintoshes. Each participant in the project downloaded a special solver binary from the network and installed it manually on his PC.

More generally, there have been proposals for global computing using low-level communication systems such as PVM[13], MPI[14], Linda[15], etc., as well as higher-level, dedicated systems, including CONDOR[7], Netsolve[6], Legion[16], and GLOBUS[17]. Although these systems offer heterogeneous collaboration of multiple systems in parallel, some of them in a Wide-area setting, they involve rather complex maintenance of different binary code, multiple execution environments, etc.

To overcome the system complexity, there have been several proposals of Java-based global computing systems, utilizing the platform independence and portability of Java. These systems are typically focused on providing the users with parallel computing capabilities with computing resources available in LAN and/or WAN.

The systems can be largely categorized into two—those that use Java Applets as computing objects, and those which do not. The former systems include JET[18], Javelin[19], POPCORN[20] and Charlotte[21]. In these systems, the idea is to implement the client's computational task as a Java Applet, and post it on a certain Web page. Then, the server provider would open the Web page, thereby downloading the Applet and executing the computation of the client.

JET is a simple system where the created Applet is merely placed on the Web page, and does not provide facilities of *guiding* the potential server providers into actually opening the page.

[2] In this case, by executing the Dispatcher on a network firewall, we can export the selected set of Dispatchers to the outside network.

In Javelin, the server provider invokes his Web browser, and opens the Broker Applet page of of Javelin (the server provide is called the *Host*). The clients, in turn, places the Applet-encoded computation and an HTML page which embeds the Applet (they are aggregately called a *task*) in his own Web server, and notifies the Broker of its URL. The Broker then places the task in the task queue, and starts scheduling the registered tasks by removing one task from the task queue and a Host from the Hosts pool, and directs the Host to open the Web page of the task. This is implemented by the Broker Applet calling the `showDocument` method of the `AppletContext`.

Javelin and Ninflet are similar in that, both systems facilitate a three-tier architecture, where the Javelin's Broker, Task and Server correspond to the Ninflet system's Dispatcher, Ninflet, and the Server. Javelin makes it easy for a end-user to act as a computing server provider by leveraging the existing Web technology, utilizing the JVM embedded in Web browsers. On the other hand, for a more dedicated clustering environment, it causes greater administrative overhead, as window systems and browsers must be invoked on every computing node. Even for wide-area use, users could easily inadvertently or deliberately point his browser to other pages, or quit the program altogether. Thus, brower-based systems such as Javelin, despite its ease of use, would result in a rather unstable computing environment, making it necessary to invest in software techniques to cope with such instability.

Also, it is not clear whether the Applet security model, which was originally designed for point-to-point downloading and remote execution, would be necessary and sufficient for global computing in a more general setting. For example, in Javelin, the Applet security model prevents direct inter-Applet stream communication, thereby requiring all inter-Applet communication to be routed through the host where the Applet class file originated, possibly becoming a significant bottleneck for parallel computing. Although recent work revealed that inter-Applet communication would be possible using RMI, it is still not clear if other security aspects would be necessary and sufficient.

We envision that, Applet-based systems such as Javelin, due to their easy participation of computing servers, would be suitable for solving a large, domain decomposible problems that arouse high public interest, such as RSA Factoring[22] and domain-decomposible type of grand challenge problems. On the other hand, they are not very appropriate for a constructing a more stable computing environments which will consist of high-performance clusters and supercomputers interconnected with high-speed WAN/LAN.

There are also other systems that are constructed as standalone Java applications, such as IceT[23]. IceT has an 'uploading' feature of classes, allowing instance creation of arbitrary classes on other IceT hosts. On the other hand, the programming model of IceT is basically an evolution of PVM, and do not account for some important issues in global computing such as automated resource allocation, fault tolerance, checkpointing, etc. Also, there are no object-oriented abstractions of parallelism control nor object distribution.

ObjectSpace Voyager[24] and Aglets[25] are so-called mobile agent systems that targets a different application area other than parallel computing, but have similar characteristics. However, such systems are not intended for efficient, global computing that require extensive communication between the objects as well as collective and coordinated resource control

6 Conclusion and Future Work

We have presented Ninflet, a Java-based global computing system which constructs a virtual computing environment in WAN/LAN settings to exploit computational resources of idle computers. The design of Ninflet emphasizes system stability, security, and fault tolerance.

The key to the success of systems such as Ninflet is how much it could capture widespread usage. One methodology is to leverage upon systems with which people feel that there is sufficient security, as is represented by Javelin and other browser-based systems. Ninflet takes a different approach, by designing a more stable system with security and other features that are tailored for global computing using Java.

We do believe both types of systems have their advantages and disadvantages. It is also possible to foresee a combined system, so that different JVMs and security models could be selected according to application requirements.

As a future work, we are exploiting the following issues:

- First, we are pursuing an effective resource scheduling algorithm for global computing. Although we have conducted extensive benchmarks in a physically networked environment[4], we are now pursuing construction of simulation model and a simulator based on the model for global computing.

- Second, we are designing libraries that implement different parallel programming styles on Ninflet. Although an earlier work investigated the possibility of library-based accommodation of different programming styles in the context of C++ on workstation/PC environments[26], we need more sophisticated support to take into account checkpointing and security issues.

- Finally, we need to develop supporting software infrastructure, such as performance, debugging, and better server management tools.

References

[1] S. Sekiguchi, M. Sato, U. Nagashima. - Ninf - : Network based Information Library for High

Performance Computing, *SIGNote of Information Processing Society*, number 94-HPC-52, pages 127–132, 1994. in Japanese.

[2] S. Sekiguchi, M. Sato, H. Nakada, and U. Nagashima. – Ninf –: Network Base Information Library for Globally High Performance Computing., in *Proceedings of Parallel Object-Oriented Methods and Applications* (POOMA), 1996.

[3] M. Sato, H. Nakada, S. Sekiguchi, S. Matsuoka, U. Nagashima, and H. Takagi. Ninf: A Network based Information Library for a Global World-Wide Computing Infrastructure, *Lecture Note in Computer Science, High-Performance Computing and Networking*, pages 491–502, 1997.

[4] A. Takefusa, S. Matsuoka, H. Ogawa, H. Nakada, H. Takagi, M. Sato, S. Sekiguchi, and U. Nagashima. Multi-client LAN/WAN Performance Analysis of Ninf: a High-Performance Global Computing System, in *Proceedings of Supercomputing'97*, 1997.

[5] H. Nakada, H. Takagi, S. Matsuoka, U. Nagashima, M. Satoh, S. Sekiguchi. Utilizing the Metaserver Architecture in the Ninf Global Computing System, submitted to *HPCN'98*.

[6] H. Casanova, and J. Dongarra. Netsolve: A Network Server for Solving Computational Science Problems. in *Proceedings of Supercomputing'96*, 1996.

[7] M. Lizkow, M. Livny, and M. W. Mutka. Condor – A Hunter of Idle Workstations, in *Proceedings of the 8th International Conference of Distributed Computing Systems*, 1988.

[8] S. Hirano. HORB: Distributed Execution of Java Programs, *Lecture Notes in Computer Science 1274 Worldwide Computing and Its Applications*, pages 29-42, 1997

[9] M. Beck, J. S. Plank, and G. Kingsley. Compiler-Assisted Checkpointing, in *25th Annual International Symposium on Fault-Tolerant Computing*, 1995.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns – Elements of Reusable Object-Oriented Software, *Addison-Wesley*, 1994.

[11] The RSA Data Security Secret-Key Challenge, `http://www.rsa.com/rsalabs/97challenge/`.

[12] DISTRIBUTED.NET – Fastest Computer on Earth, `http://www.distributed.net/`.

[13] V. S. Sunderam. PVM: A Framework for Parallel Distributed Computing. *Technical Report ORNL/TM-11375*, Department of Math and Computer Science, Emory University, 1990.

[14] MPI: A Message-Passing Interface Standard. The International Journal of Supercomputer Applications and High Performance Computing Systems, 1988.

[15] R. A. Whiteside and J. S. Leichter. Using Linda for Supercomputing on a Local Area Network. *Technical Report YALEU/DCS/TR-638*, Department of Computer Science, Yale University, 1988.

[16] A. S. Grimshaw and W. A. Wulf. Legion: Flexible Support for Wide-Area Computing, in *Proceedings of the Seventh ACM SIGOPS European Workshop: Systems Support for Worldwide Applications*, 1996.

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

[18] L. M. Silva, H. Pedroso, J. G. Silva. The Design of JET: A Java Library for Embarrassingly Parallel Applications, in *Proceedings WOTUG'20 – Parallel Programming and Java Conference*, IOS Press, pages 210–228, 1997.

[19] B. O. Christiansen, P. Cappello, M. F. Ionescu, M. O. Neary, K. E. Schauser, and D. Wu, Javelin: Internet-Based Parallel Computing Using Java, *ACM Workshop on Java for Science and Engineering Computation*, 1997.

[20] The POPCORN Project, `http://www.cs.huji.ac.il/~popcorn/`

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

[22] RSA Factoring-By-Web project!, `http://www.npac.syr.edu/factoring.html`.

[23] P. A. Gray, and V. S. Sunderam, The IceT Environment for Parallel and Distributed Computing, in *Proceedings of 1997 International Scientific Computing in Object-Oriented Parallel Environments Conference*, 1997.

[24] ObjectSpace Voyager – The Agent ORB for Java, `http://www.objectspace.com/voyager/`

[25] Programming Mobile Agents in Java, `http://www.trl.ibm.co.jp/aglets/`.

[26] S. Matsuoka, A. Nikami, H. Ogawa, and Y. Ishikawa. Towards a Parallel C++ Programming Language Based on Commodity Object-Oriented Programming, in *1997 International Scientific Computing in Object-Oriented Parallel Environments Conference* (ISCOPE'97, `http://www.acl.lanl.gov/iscope/`), Springer LNCS 1343, Marina Del Rey, CA, Dec. 1997.