**Project Description**

Our project is a distributed computing framework, in the same vein as SETI@Home, condor, grid computing, and boinc.

**Project Goals**

The goal of our project was to provide a framework for distributing work from one machine to other available machines. Another goal was to make work tasks generic enough that any of the available machines would be able to run them; they need not know anything about the work they perform. Finally, we did not add every possible feature, but we hoped to provide a solid enough framework of useful features that it could easily be built upon.

**Implementation Approach**

To accomplish our goals, we divided the project into four entities:

1) **Work Provider**: The Work Provider has tasks that it wants to distribute to available Work Engines.

2) **Work Engine**: The Work Engine registers with the Work Coordinator. Next, it accepts and performs tasks assigned by authorized Work Providers. When it has finished the work, it returns the results to the Work Provider. The Work Engine is just a provider of processing power; it knows nothing about the task that it performs.

3) **Work Coordinator**: The Work Coordinator records which Work Engines are available to perform tasks, and gives this information to Work Providers that request it.

4) **Tasks**: A Task is any chunk of work provided by a Work Provider to be done by a Work Engine. They are in the form of a generic serialized object; the Work Engines can do any Task given to them and need know nothing about it.

**Scope**

The range of problems our project can solve includes both Non-Distributable Problems and Distributable problems.
A Non-Distributable Problem is one where the problem may not be broken down into multiple pieces, and so there is only one piece of work to give out. An example of a Non-Distributable Problem is calculating Pi. This calculation can be done once by a Work Engine, with one final answer.

A Distributable Problem is one where the problem may be broken down into multiple pieces, and so can be distributed to more than one computer to work on simultaneously. An example of a Distributable Problem is calculating Happy Numbers. The problem is searching for Happy Numbers within a range of numbers. There may be infinite Happy Numbers found within infinite number of ranges. Each range may be given out to a separate Work Engine, with no dependency between the ranges or other Work Engines’ results. This problem is easily distributable.

Outside Our Scope

A few things that we did not address within this project were:

1) Load Balancing: We did not attempt to spread the requests evenly among the Work Engines. The same Work Engines may keep getting requests, while others are idle.

2) Fairness to Work Providers: We do not queue Work Provider requests. Whichever Work Provider requests a Work Engine first receives it.

3) Reward/Payment: There is no reward for Work Engines completing tasks. Including a reward system would allow a user to model real world problems with cheating.

Implementation Details

The framework we have built is written completely in Java. This gives it platform independence, allowing components to run on different hardware and operating systems without needing to be recompiled.

The communications infrastructure is built using Java’s RMI technology. RMI handles most of the nitty-gritty details of communicating between machines, abstracting everything to simple method calls. The only downside to RMI, if it could be considered one, is that there is a slight overhead in terms of the complexity of the deployment. There needs to be an RMI registry available for each component, some command line
arguments must be specified, and a security policy must be present for each component to
run. This is not all bad though, as these extra requirements actually represent additional
features that we could make more use of if needed.

The functionality of the coordinator, engines, and tasks is defined in three interfaces:
ComputeCoordinator, ComputeEngine, and Task. These interfaces define the messages
that pass between the components and the individual component responsibilities. There is
actually a great deal of flexibility in our framework in that it supports any implementation
of the engine (and coordinator) as long as it conforms to the responsibilities outlined in
the interface. To date, we have only a single implementation of the engine called
SimpleEngine, but the capability is there to provide addition implementations that could
have features such as limiting processing time to periods when the CPU is not busy doing
other things.

The ComputeCoordinator is responsible for tracking available ComputeEngines and
providing information about them to interested providers. Its interface is actually quite
simple, and looks like this (with comments and Exceptions removed for brevity):

```java
public interface ComputeCoordinator extends Remote {
    public void registerAsAvailable(EngineId id);
    public void registerAsUnavailable(EngineId id);
    public Collection getAvailableClients();
}
```

As you can see, there isn’t much to it. ComputeEngine’s must register themselves and
provide an EngineId with identifying info such as the URL where they can be reached.
You can’t see it here, because the comments have been removed, but the interface
actually specifies that the engine must re-register itself every 10 seconds. This heartbeat
mechanism prevents the coordinator from keeping stale information and giving it out to
Providers.

The ComputeEngine interface is almost equally simple (again with comments and
Exceptions removed):

```java
public interface ComputeEngine extends Remote {
    public boolean reserveEngine(ProviderId providerId,
                                  ReservationToken token);
    public boolean unreserveEngine(ProviderId providerId,
                                  ReservationToken token);
    public Object performTask(ProviderId providerId,
                                  ReservationToken token,
                                  Task task);
```

The provider must reserve the Engine after it has gotten information about it from the coordinator. Then it is free to send it a task. If it chooses not to send it a task after reserving it, it is responsible for calling the unreserveEngine method. We chose to implement the engine reservations at the engine itself instead of the coordinator because we felt that this provided the most robustness with the least cost. The engine always knows for certain whether or not it is available while the Coordinator can potentially get out of sync.

You’ll also note the presence of the ReservationToken in these method calls. Each engine generates a single use token that is passed to the coordinator as part of the EngineId. This token can be redeemed to get a single task performed by the Engine. Once a token has been redeemed once, it is no longer valid. This mechanism prevents a provider from getting information about an engine and then monopolizing the engine with repeated tasks. Its token will expire after the first task so the provider must re-acquire the engine through the coordinator before it can use it again. This allows the coordinator to enforce some fairness protocols if so desired.

To ask the engine to do its actual work, the provider calls the performTask function and passes it a Task object. The Task itself is the simplest interface:

```java
public interface Task implements Serializable {
    public Object execute();
}
```

consisting of a single execute method that returns an Object. This allows the providers great flexibility in writing tasks. We have implemented sample tasks that do basic mathematical functions such as calculating Pi or finding “Happy Numbers”, but there is nothing to prevent more complicated tasks such as ray-tracing single frames in an animation sequence from being implemented. In fact, there is nothing that explicitly prevents a Task from doing callbacks to its provider during its execution or even communicating with other Tasks on other Engines. The only limitations placed on tasks are created by the security policy file. The policy file may limit file writes or network connections by tasks configured on a host-by-host basis.
The three interfaces presented here define the interaction points of the various components of our system. Together they define a system that allows tasks of any type to be seamlessly distributed amongst as many machines as desired.

**Tasks**

For our project we created 3 tasks to use in our framework. A Happy Numbers Task, Kaprekar Numbers Task, and an n-digits of Pi Task.

The Happy Numbers Task calculated which numbers in a given block were Happy and Unhappy. Here is the definition of a Happy Number (from mathworld.wolfram.com):

Let the sum of the squares of the digits of a positive integer $s_0$ be represented by $s_1$. In a similar way, let the sum of the squares of the digits of $s_1$ be represented by $s_2$, and so on. If $s_n = 1$ for some $n$, then the original integer $s_0$ is said to be happy.

An example would be the integer 7. 7 squared is 49. Continuing along we get 49, 97, 130, 10, 1. The sequence terminates at 1 so 7 is a happy number. If it is unhappy, it continues to loop through a subsequence and never reaches 1. For example 4. 4, 16, 37, 58, 89, 145, 42, 20, 4.

The Kaprekar Numbers Task calculates which numbers in a given block are Kaprekar numbers. Again from mathworld.wolfram.com is the definition of a Kaprekar Number:

Consider an $n$-digit number $k$. Square it and add the right $n$ digits to the left $n$ or $n-1$ digits. If the resultant sum is $k$, then $k$ is called a Kaprekar number.

Examples of Kaprekar numbers are 9 and 297. $9^2=81$; $8+1=9$. $297^2=88,209$; $88+209=297$.

The n-digits of Pi Task simply calculates $\pi$ to $n$ decimal places. It isn’t really distributable, it was just borrowed from Sun’s Java site as an example of another type of Task for our framework.

**Results**

Following are some results that we collected:

Times were taken inside the Java environment so as not to count any JVM startup time.
Finding Happy Numbers 0-1,500,000 (chunks of 500,000)
distributed among 3 clients: 40 seconds
serially on one computer: 1 minute 1 second
one chunk on one computer: Java out of memory error after ~ 9 _ minutes

Finding Happy Numbers 0-3,000,000 (chunks of 500,000)
distributed among 6 clients: 1 minute 7 seconds
serially on one computer: 2 minutes 4 seconds

Here we can see a clear difference between running the HappyNumber task in a distributed environment and running it in a non-distributed environment. When distributed, the Happy Numbers can be computed much more quickly.

Finding Kaprekar Numbers 0-1,500,000 (chunks of 500,000)
distributed among 3 clients: 15 seconds
serially on one computer: 40 seconds
one chunk on one computer: 39 seconds

Finding Kaprekar Numbers 0-3,000,000 (chunks of 500,000)
distributed among 6 clients: 16 seconds
serially on one computer: 79 seconds
one chunk on one computer: 79 seconds

Again we can see a clear difference between distributed and non-distributed. In the Happy Number scenario the difference isn’t as dramatic as it is here since it spends a lot of time transferring the final results back to the provider. With Kaprekar Numbers however, the numbers are much sparser. There isn’t so much data to transfer back at the end so we can end up with calculating twice as large of a number space, but only increasing the time spent by one second.

Comparing the Kaprekar Numbers and Happy Numbers we also see another difference in that the Happy Numbers calculation runs out of memory when run as a single large task. So not only does distributing tasks help speed but it also helps memory usage which is especially significant in some situations as shown above.

With the Pi Task we borrowed from Sun’s Java site, it wasn’t a distributable task so results aren’t really interesting. It takes the same amount of time to compute locally as it does remotely besides less than a second of communication overhead.