CIS630: Distributed Computing

Shad Stafford, Shelby Stair, Toby Ehrenkranz
Outline

- Project Description
- Project Goals
- Implementation Approach
- Scope
- Implementation Details
- Results
Project Description

- Build a distributed computing framework
- Project inspired by existing distributed computing projects such as:
  - seti@home
  - condor
  - grid computing
  - boinc
Project Goals

- Want to be able to distribute work from one machine to one or many other machines
- Work units should be generic, able to do any kind of task
- No bells and whistles to start with, but with a solid enough framework they could be added
Implementation Approach

- Three entities:
  - Work Provider: Has work to pass out to available Work Engine(s)
  - Work Engine: Can do work received from Work Providers
  - Coordinator: Records which Work Engines are available to receive work, and informs Work Providers about them
Implementation Approach

- Tasks
  - Tasks are any chunk of work provided by a Work Provider to be done by a Work Engine.
  - Tasks are generic serialized objects that any Work Engine can do any type of Task.
Scope: Solvable Problems

- Non-Distributable Problems
  - A single chunk of work that can be given to an idle machine to do
  - Example: Pi
    - Solved Once

- Distributable Problems
  - A problem that can be broken into many pieces and given out to many idle machines
  - Example: Happy Numbers
    - Each participating Work Engine gets a range of numbers to work on
Scope: Outside the Scope of this Project

- No load-balancing
  - Same Work Engines at the top of the list could be used over and over

- Only somewhat fair
  - Requests are not queued by the Coordinator.
  - Whoever requests an available Work Engine first gets it.

- No payment/reward for doing work
  - Not modeling real-world problems of cheating
Implementation Details

- All components implemented in Java
- RMI is used for communication layer
  - RMI handles lots of details for us
    - Multi-threaded engines so can respond to other requests while processing work
    - Our code doesn’t need to worry about sockets
- Dynamic class loading
  - Work unit can be any kind of object as long as it is serializable and implements the Task interface
Communication

- Message Types
  - Engine tells Coordinator available/not-available
  - Provider requests list of available Engines
  - Provider contacts Engine to reserve it
  - Provider gives Engine a Task to do
public interface ComputeCoordinator extends Remote {
    public void registerAsAvailable(EngineId id);
    public void registerAsUnavailable(EngineId id);
    public Collection getAvailableClients();
}
Compute Engine Interface

```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);
}
```
Task Interface

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

- Engines provide heartbeat messages to the coordinator
  - Coordinator drops stale engines
- Engine reservation is for one-time task
  - Engine provides single use token to coordinator which is distributed to providers
  - Provider can use Engine once, but then must re-acquire through coordinator
    - Prevents monopolizing of engines
    - Allows for more sophisticated load balancing/fairness
Experimental Results

- Talk about the actual providers that we implemented: happy, kaprekar, pi. Give description of exactly what happy and kaprekar numbers are.
- A couple slides showing that our framework really does work properly.
😊 Happy Numbers 😊

- 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.

- Ex: 7, 49, 97, 130, 10, 1
Happy Numbers Runtimes

- 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
Kaprekar Numbers

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.

Ex:

- \( 9^2 = 81, \ 8+1 = 9 \)
- \( 297^2 = 88,209, \ 88+209 = 297 \)
Kaprekar Numbers Runtimes

- 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
Pi Task

- Calculates \( n \) digits of pi.
- Not really distributable, just a task we borrowed from Sun’s Java site for another example of a Task.
- Running locally and remotely only differ by less than a second.
Beautiful screenshots.
Admit you like the screenshots.
Okay, the screenshots suck. 😞