SMALL WEB COMPUTING: DISTRIBUTING COMPUTATIONS ACROSS THE INTERNET

First Designed by David Arnow and Kevin Ying, CUNY, 2000

Redesigned and Extended by Paula Whitlock, Dino Klein and many others, 2000-2003

Why is another distributed software package needed?

Description of the original software and recent developments

Future evolution of the software
Parallel and Distributed Computing:

SMP multiprocessor (Sun Multiprocessor, IBM, Dell):

   Operating Systems dependent:
      User programs with fork() - Unix, Linux, most MS OS
      threads - Unix, most recent Linux

   Use a parallelizing compiler - concurrent C, parallel Fortran,
      parallel Haskell

Distribute the calculation over a local network -

   Use a message passing library - PVM, MPI, DP - all use remote
      procedure calls - Unix, most recent Linux
      User responsible for all remote processes.

Perform the calculation in a cluster environment -

   Beowulf clusters - provides software to enable easy distribution
      of the calculation, must use PVM or MPI

   Openmosix clusters - OS distributes the calculations, user must
      divide calculations with fork(), threads not supported

Distribute calculations over the internet -

   User provides software for download - individuals obtain the software
      and execute it on their own computers

Developing a parallel/distributed computation involves considerable effort!
Simple Web Computing - goals

a) easy user interface - a design that separates the programming interface from the underlying architecture.

b) embrace all three paradigms described above - master-worker MIMD parallel programming model

c) provide some security guarantees - uses the "sandbox" security model offered through the use of java applets over the internet

d) extendable software - easy to add new features to the software
SWC is a small, simple framework, originally developed as a tool for students, that simplifies developing and deploying WebComputing applications. The objective of the SWC system is to provide a layered WebComputing system design that separates the programming interface from the underlying WebComputing architecture. Figure 2 illustrates the SWC architecture. The large middle box represents the SWC system proper and contains two levels. The upper level, shown in the dashed box, provides the SWC programming API. The lower level provides the support for this API. This design approach shields application developers from the underlying dynamic and unreliable execution environment and makes the application programs portable, yet at the same time provides system developers freedom in architecture design.
Because of its independent architecture design, SWC is able to provide three implementations of its architecture (see Figure 4). One is thread-based and runs on an SMP platform; it serves as a development environment when a web-based one is not available or is inconvenient. Another implementation is based on independent Unix processes. The third implementation serves our main purpose — WebComputing. It consists of a multithreaded, servlet-enhanced HTTP server that provides an application control page, creates the master process, downloads the applets and handles all communication. It also provides the classes that define, for both the master process and the applets, their computational structure and their communication tools.

An SWC computation is initiated using a form in a web page provided by the server. A servlet responds by creating the necessary objects within the HTTP server along with an external master process. The master process and the server communicate using the connection-oriented TCP protocol and need not reside on the same machine. The master process, using programmer provided classes creates an initial set of task definitions, which are sent off to the server. The server maintains a collection of task definitions, and uses eager scheduling, as is commonly done in WebComputing [AISS,97] [BKKW,96] [CCINSW,97], to assign them to applets that have been downloaded into volunteering web clients. Because of the unreliable nature of the applets themselves, the fact that the most obvious candidates for WebComputing will not require large quantities of data to define tasks, and the desire to eliminate system-imposed limits on the number of connections on the server as a potential bottleneck, plus the performance consideration [YAC,99], server-applet communication uses the connectionless UDP protocol. Because the server does not consider a task complete until the results are actually received, lost packets do not compromise the integrity of the application. To avoid failure to utilize an applet as a result of communication failure, applets use a timeout mechanism to repeatedly send the server their most recent response until the server provides additional work or instructs them to terminate.

As the server receives responses from applets, it determines whether these are additional task definitions, in which case they are added to its collection of task definitions, or results from completed tasks in which case they are passed to the master. The latter may, in response, generate new task definitions or control information for the server to distribute to the applets. Control information is immediately broadcast to applets and made available to all future ones. Figure 5 illustrates SWC communication between different modules.
SWC is a small, simple framework, originally developed as a tool for students, that simplifies developing and deploying WebComputing applications. The objective of the SWC system is to provide a layered WebComputing system design that separates the programming interface from the underlying WebComputing architecture. Figure 2 illustrates the SWC architecture. The large middle box represents the SWC system proper and contains two levels. The upper level, shown in the dashed box, provides the SWC programming API. The lower level provides the support for this API. This design approach shields application developers from the underlying dynamic and unreliable execution environment and makes the application programs portable, yet at the same time provides system developers freedom in architecture design.
3. Performance

The efficiency of a WebComputing application depends on many elements beyond the control of the implementer of a WebComputing system or framework. Among these are the appropriateness of the application to the platform because of the extreme network latency and bandwidth limitations of the Internet, the implementation of the Java Virtual Machines (JVM) that support the applets, and the network conditions that affect latency and bandwidth. To provide a sensible estimation of the system’s performance, we need to quantify the communication overhead. Data transfers between different system modules constitute the dominant portion of the WebComputing system overhead. Data flow of the SWC system can be classified into three categories:

- **MW**: The SWCMaster sends newly defined tasks to a SWCWorker.
- **WM**: The SWCWorker sends SWCResultUnits to the SWCMaster.
- **WW**: The SWCWorker sends SWCWorkUnits to the SWCRouter for rescheduling to another SWCWorker.

Evaluating the communication costs for these three categories can provide insight to the system's behavior. A WebComputing application developer may use this information and the communication pattern of the specific application to estimate its communication overhead.

To study these costs, performance measurements were carried out using JDK's Solaris Production Release version 1.1.7 and its default just-in-time (JIT) compiler, on two Sun Ultra-Sparc-10 workstations with 128 MB of RAM each, running Solaris 7 operating systems. One machine was employed as a server, the other as a client. These machines were connected with 100Mbit Ethernet LAN. The network latency was below 1 millisecond (ms) (tested with UNIX ping facility). We also used Netscape Communicator 4.51 with a 1.1.5 Java run-time environment when applet implementation of the SWC system was tested.

Table 1 shows the communication costs for different categories of application data flow using an object with a 45-byte internal state. For both thread and process measurements, SWCMaster, SWCWorker, and SWCRouter ran on a single workstation as three threads and three processes respectively. For the Web Applet measurement, two workstations were employed. Both SWCMaster and SWCRouter ran on the server as
Characteristics of the SWC software:

- Written in Java
- Provides a set of extendable classes that provide the framework for the computations:
  - define a particular task or result,
  - a mechanism for recognizing task equivalence
  - a mechanism for recognizing task completion
- Upper level communication (master process and server) uses TCP, lower level communication uses UDP
- Eager scheduling used to guarantee task completion and load balancing

Worked well on small problems, decided to apply the software to a much larger simulation

Difficulties:

1. Using UDP for applet-router communication limited the amount of data that could be transmitted easily by the user - maximum work unit size was limited to 64KB as no fragmentation/reassembly code was present
2. Using UDP required code to guarantee delivery of data and results. This additional code became a bottleneck and source of "bugs"
3. The Java framework was outdated
SWC 2:

TCP: The new system relies on TCP exclusively for both upper level and lower level communication.

- TCP simplified programming - no need to include testing and retransmission of lost results
- TCP allows transfer of large data sets (no restriction on data or results size)
- Testing showed that using TCP was not a bottleneck. Both data set size and frequency of communication were tested.

Computation Specificity: Under SWC, every problem required specific master, router and worker processes. The overhead in running multiple computations could be quite large.

- Under SWC2, the main server can coordinate multiple master processes.
- The SWC2 system reuses the same routers and workers for multiple computations. This increases efficiency and centralizes control.

Communication model: SWC required router-to-router communication

- SWC2 relies on the pull model - all communication is initiated by a lower-level player to an upper level, e.g. worker-to-router, router-to-master.
Application of the SWC2 software to a large computation - a Monte Carlo simulation of the properties of hard hyperspheres in many dimensions (Marvin Bishop, Manhattan College)

Characteristics of the simulation:

- The properties of the system of hard hyperspheres are calculated as expectation values:

\[
<A> = \frac{\int A(R) f(R) \, dR}{\int f(R) \, dR}
\]

The multidimensional function, \(f(R)\), was sampled using the Metropolis, \textit{et al} algorithm and the resulting points, \(R_i\), used to evaluate \(A(R)\). This estimator was used to calculate \(<A>\)

- The random walks used to sample \(f(R)\) were straightforward to divide between multiple instances of the same computation.

- As the number of dimensions increases, the time it takes for the random walks to converge to their asymptotic distribution increases significantly

SWC2 task - follow N Metropolis random walks for X steps

Is it easy to use? Must include the appropriate SWC2 classes in the Java software.

1. The master process must implement a Computation, in it a communication object is defined:

```java
public class ParticleComputation implements Computation {
    // define the WorkQueueAccess object and needed variables
    WorkQueueAccess wqa;
    double density;
    int dimensions;
    double[] g;
    int workGotBack = 0;
    int workDistributed = 0;
    ...
}
```
The method loadConfiguration() receives the parameters for the calculation through the Map object named config, receives the object WorkQueueAccess and receives an output stream where it can record various information.

```java
public void loadConfiguration (Map config, WorkQueueAccess wqa, OutputStream os) throws Exception {
    this.wqa = wqa;
    this.config = config;
    // read in the number of dimensions
    dimensions = getInt("dimensions");
    // read in the density of the system
    density = getDouble("density");
}
```

The method run() instantiates the class ParticleWorkUnit which extends WorkUnit, sends out the work units in addWorkUnit() and receives back the completed result units in getResultUnit().

```java
public void run() {
    for (i =0; i < numTasks; i++) //numTasks is the number of work units
    {
        ParticleWorkUnit wu = new ParticleWorkUnit();
        wu.WorkUnitID = workDistributed;
        wu.WorkerClassName = "particle2.ParticleWorker";
        wu.dimensions = dimensions;
        wu.density = density;
        try
        {
            wqa.addWorkUnit(wu);
        }
        catch (Exception e)
        {
            ParticleResultUnit sru;
            while (workGotBack<workDistributed)
            {
                try
                {
                    sru = (ParticleResultUnit)wqa.getResultUnit();
                }
                catch (Exception e)
                {
                }
            }
        }
    }
}
```
The computation is completed when all the result units from the workers are combined into the final averages, the statistical errors are calculated and the results are written.

2. A separate class implements the worker code

```java
public class ParticleWorker implements Worker {
    double density;
    ParticleWorkUnit pwu;
    ParticleResultUnit pru;
}
```

A work unit is assigned in the method setWorkUnit().

```java
public void setWorkUnit (WorkUnit wu) {
    pwu = (ParticleWorkUnit)wu;
}
```

The run method extracts the task parameters and performs the calculation.

```java
public void run() {
    dimensions = pwu.dimensions;
    density = pwu.density;
    g = new double[ngr];  //ngr is the number of values of g(r)
    pru = new ParticleResultUnit(pwu);
    pru.g = g;
}
```

The ResultUnit is extracted and returned upon completion of the task.

```java
public ResultUnit getResultUnit () {
    return pru;
}
```
How well does it work?

5-dimensional simulation of 3125 hyperspheres

- Serial C++ code on a single processor of a Sun Enterprise - 21,000 random walk steps in 34 hours
- SWC2 code, 10 threads on a 14 processor Sun Enterprise - total of 30,000 random walk steps in 9 hours and 26 minutes
- SWC2 code, 10 processes distributed over 10 SunBlade computers on a college LAN - total of 30,000 random walk steps in 9 hours and 56 minutes
- SWC2 code, 10 applets distributed over the internet - the volunteer computers were a PC running Solaris 5.8 with Netscape 4.7, a laptop and a Pentium 3 running Linux 2.4 with Mozilla 1.0.1 and a PC running Windows 2000 with Netscape 6 or Internet Explorer 5.5. The server was on a Pentium 3 running Linux 2.4.

All the calculations gave the same results within statistical errors.
Future Directions:

- Communication issues:

  SWC2 works well for calculations that can be divided into independent tasks and all communication is between levels.

  Many calculations require communication of interim results between workers - this feature has to be incorporated into the software.

- Robustness of the software:

  SWC2 supports multiple routers so that if one router is terminated or the processor fails, workers can return their results to an alternative router.

  However, if the single server fails, the whole computation is lost. A multi-server model needs to be developed and implemented.

- Language Specificity:

  All calculations that use SWC2 must be written in Java.

  Experiment with combining the an envelop of Java with functions written in other languages.

  Experiment with conversion software that takes a program written in one language and converts it to Java.