Technical University Berlin

Telecommunication Networks Group

The ns-2/akaroa2-project

Andreas Köpke, Hagen Woesner

koepke@ee.tu-berlin.de, woesner@ee.tu-berlin.de

Berlin, July 2001

TKN Technical Report TKN-01-008

TKN Technical Reports Series

Editor: Prof. Dr.-Ing. Adam Wolisz

Abstract

This report describes the combination of ns-2 and Akaroa-2. Within the first chapter, both tools are introduced shortly. It is assumed that the reader is familiar with ns-2 already to a certain point at which the results of the simulation can be drawn out of the simulation model. It is not required that the reader is familiar with Akaroa-2, although a brief look into the manual of it is strongly recommended. After that, the combination of both tools is emphasized along with the problems that had to be solved. A detailed instruction how to install the interface is then followed by an example that shows how to obtain the mean length of a M/M/1 queue. The source code of the C++ and OTcl classes that had to be changed or invented concludes this report.

Contents

Chapter 1

Introduction

1.1 Motivation

Within the last decade, discrete simulation has become a standard tool of scientists and engineers wherever it was necessary to estimate the behavior of large stochastic systems like computer networks or national economies. As can be seen in the number of contributions to scientific journals or conferences, almost every result that has been derived analytically is nowadays justified or verified by simulation curves that support the thesis of the author.

Although quantitative stochastic simulation is a useful tool for studying performance of stochastic dynamic systems,it can consume much time and computing resources. To overcome these limits, parallel or distributed computation is needed.

Universities and research institutes often have lots of computers connected in a LAN. Using these heterogenous (in terms of speed) computers for simulations is a straightforward idea. One of the main obstacles is the easy distribution of simulations over this cluster. There are two approaches to solve this problem. One is the explicit parallelization of the simulation and the other one is the MRIP-approach taken by Akaroa-2. It runs *Multiple Replications In Parallel*. This results in an almost linear speedup with the number of hosts.

NS-2 is a nice tool for network simulation, but does not provide support for statistical analysis of the obtained results.

So by combining NS-2 and Akaroa-2 we add run-length control for simulations based on statistic measures to NS-2 as well as a speed-up if the simulation can be run on many hosts in parallel. The existing package for writing parallel simulations with ns is – at least in our eyes – a bit more complicated and does not provide statistical run length control.

The changes were made only in the NS-2 package. The original code seems to be quite stable, so these changes should not depend too much on the NS-2 version used.

1.2 NS-2

NS-2 is a discrete event simulator targeted at networking research. It provides substantial support for simulation of TCP, routing, and multi-cast protocols over wired and wireless (local and satellite) networks. Simulations are written in OTcl but if you need some specific behavior of the OTcl Classes you can write your own in C++. For more information about

Network Simulator check the home page of the ns project [3]. For a good tutorial refer to [4]. A Tcl/Tk- tutorial can be found at [5].

Ns-2's main advantage is the multitude of network protocols implemented. On the other hand it lacks support for statistical evaluations. Usually one writes the interesting variables into a trace file, and measures such as mean and variance are evaluated with an awk-script. But how many values should be written into the trace file? Sometimes simulations are run much longer than necessary – or more often much shorter, which devaluates the conclusions drawn out of the simulations. To get an expression of the quality of any simulation or measurement result Another problem concerns the length of the random number generator. Some simulations need less than a day to exhaust the random number stream.

1.3 Akaroa-2

Akaroa-2 is designed for running quantitative stochastic discrete-event simulations on Unix multiprocessor systems or networks of heterogeneous Unix workstations. Instead of dividing up the program for parallelization – a tremendous effort – multiple instances of an ordinary simulation program are run simultaneously on different processors.

For more information about Akaroa check the home page of the AKAROA-2 project [6]. Akaroa-2 is a process oriented simulation engine. Written completely in C++ it is easy to write fast simulations with it. The random number generator used in the current release has a really long period. It would take decades to exhaust it. The main disadvantage is the lack of network protocols. It uses statistical run length control, and the observed variables are summarized in two statistics, mean and variance, which is enough for most purposes. If you need some more you can write your own class.

Akaroa-2 does the parallelization of the simulation in a client-server manner. A unique process called `akmaster` runs on one machine in the cluster and controls the execution of the simulations using a number of `akslave` processes. These have to be started beforehand on every machine that shall be used to run simulations on it. The command to run the simulations is e.g.:

```
> akrun -n 5 ns mm1.tcl
```

The value of n=5 means to start the simulation on five machines, whereas the simulation to be executed is given after. Eventually, a result should appear in the form of:

Param	Estimate	Delta	Conf	Var	Count	Trans
1	97.6467	0.400812	0.95	0.0405465	264078	1259
2	2.77603	0.134357	0.95	0.0045561	263346	1673

Chapter 2

Statistical Security

In order to ensure that the made predicates have a statistical security, it is necessary to control the temporal duration of the simulation. For this purpose AKAROA-2 records results and calculates the half–length H of the confidence interval. The half–length is the momentary deviation from the current average value.

Given a number of values x_n then the mean value is:

$$\bar{x}(n) = \frac{\sum^n x_i}{n} \tag{2.1}$$

and the variance of the mean values $\bar{x}(n)$ is [**?**]:

$$\sigma_n^2 = \frac{\sum(x_i - \bar{x}(n))^2}{n-1} \tag{2.2}$$

Thus the half–length H of the confidence interval, which indicates the precision of the mean value, which is to be situated within statistical security z,

$$H = z \cdot \sqrt{\frac{\sigma_n^2}{n}} \tag{2.3}$$

whereby H is situated in both negative and positive direction of the mean value $\bar{x}(n)$.

$$[\bar{x} - H, \bar{x} + H] \tag{2.4}$$

In the above notation σ_n^2 is the variance and n the number of simulation values. The value z corresponds to a certain safety limit of the confidence interval that can be stated. The corresponding confidence level values of z can be taken from a table [**?**].

confidence level	z
0,90 %	1,645
0,95 %	1,966
0,99 %	2,576

Chapter 3

Interface internals

During the project we experimented with several implementations. The first one was the most conservative and was guarded against a lot of errors that simply don't occur. Although this was useful to learn about the structure of NS-2 and Akaroa-2 the overhead was unnecessary and following versions were much simpler.

We ended with a new file `akaroa.cc` and some changes in the existing files of NS-2 `rng.h` and `rng.cc`. You will have to install all these files in order to use the Akaroa-NS-interface.

3.1 Call mapping

In the file `akaroa.cc` the complete Akaroa-NS-interface is defined. The interface consists of the new class `Akaroa`, which is derived from `TclObject`. The class `AkaroaClass` provides the interface for Tcl to create C++-objects. When an `Akaroa`-method is called from OTcl, the complete OTcl string is passed to the `Akaroa`-method `command`. In this method the string is evaluated by simple string comparisons and the appropriate library function is called.

3.2 Random Number Generator

When running multiple replications of simulation model in parallel, it is important that each simulation engine uses a unique stream of random numbers. So we had to change the Random Number Generator.

The new class `AkRNGImplementation` was derived. It is also contained in the file `akaroa.cc`. It maps calls for a new uniformly distributed random number to the Akaroa Random Number Generator. Additionally some initializations are performed.

The original `RNGImplementation` of NS-2 was not intended to be inherited, so changes `rng.h` and `rng.cc` became necessary.

The class `RNGImplementation` has now a `virtual long next()` and a `virtual long next_double()` methods. This class and `AkRNGImplementation` don't belong to the public interface, but are used only internal. The "official" interface to the random number generator is the class `RNG`. We added the `static void setRNGImplementation(RNGImplementation *imp)` method, to set a new RNGImplementation when needed. The `Akaroa`-class uses it to install its own implementation. Due to its dynamic nature the random number stream of `RNG`

is a pointer now. We added reference counting for the `stream` to ensure that there is only one `RNGImplementation`-object.

It is impossible to use more than one random number stream, even multiple `RNG`-objects would use only one. This is a restriction if you want to use NS-2 without `Akaroa`.

The seeds which can be set and obtained are useless with `Akaroa`. Seeds are managed centrally by the `akmaster`-process.

Chapter 4

Installation

4.1 Preparation

- get Akaroa and install it (might need some tweaking to run under Linux)
- get Ns-2 and install it
- get files and install changed `rng.h`, `rng.cc` and new `akaroa.cc` into your `/wherever/ns/ns-2.1b7` directory

4.2 Putting things together

- Changes in the `Makefile.in` of ns:
  - approx. l. 59: set an include directory pointing to the header files (`akaroa.H`) of your akaroa installation
  - approx. l. 68: set a lib directory pointing to the akaroa libraries `-L/wherever/akaroa/lib -largs -lakaroa`
  - in the same line, add -lfl (akaroa needs the flex-library)
  - approx. l. 170: add `akaroa.o` to the objects list
- run `./configure`
- run `make` etc. as described in the ns-Manual.

Chapter 5

Usage

5.1 From OTcl

In your `ns-simulation.tcl` file do the following:

1. instantiate a new Akaroa object in the Tcl script method

   ```
   set ak [new Akaroa]
   ```

2. if you want to observe more than one parameter, tell Akaroa about it, else leave the following line out:

   ```
   $ak AkDeclareParameters $number-of-parameters
   ```

3. inform Akaroa about each parameter change

   one: `$ak AkObservation $delay`

   more:
   ```
   $ak AkObservation 1 $delay
   $ak AkObservation 2 $packet-size
   ```

5.2 From C++

Since Akaroa is written in C++, we did not write a special interface. So if you want to observe a parameter in your C++-Class, write some interface code for OTcl. During the OTcl setup you should tell your class, which variable to observe as well as its Akaroa number. When running your simulation your class should directly call the Akaroa library functions. Anything else would be too slow.

This is slightly abstract, but follows the ns philosophy closely: use Tcl for setup stuff and C++ for everything when time matters. Akaroa is unable to tell whether the library function was called by the Akaroa class or whether it is merely a library function call. These ways are completely equivalent – except for speed.

Bibliography

[1] Pawlikowski, K., Yau, V. and McNickle, D., *Distributed stochastic discrete-event simulation in parallel time streams*, Proc. of the 1994 Winter Simulation Conference, pp. 723–730, 1994.

[2] Pawlikowski, K., McNickle, D. and Ewing. G., *Coverage of confidence intervals in steady-state simulation*, Journal Simulation Practice and Theory, 6(1998), pp. 255–267.

[3] The home page of the ns project: http://www.isi.edu/nsnam/ns/

[4] Marc Greis' ns Tutorial: http://www.isi.edu/nsnam/ns/tutorial/index.html

[5] Tcl/TK Tutorial page: http://hegel.ittc.ukans.edu/topics/tcltk/

[6] AKAROA-2 home page:
http://www.cosc.canterbury.ac.nz/research/RG/net_sim/simulation_group/akaroa/about.chtml

Files

.1 akaroa.cc

```
/*
 * File: Code for a Akaroa class for the ns
 *       network simulator
 *
 *
 */

#include <stdlib.h>
#include "akaroa.H"
#include "tclcl.h"
#include "rng.h"
#include <errno.h>
// Akaroa Objects
class Akaroa : public TclObject {
 public:
  Akaroa();                // Constructor
  // Method for dynamic binding of Calls
  int command(int argc, const char*const* argv);
 private:
  // Helper functions for conversions, for error handling.
  int convert_to_int(const char *const ctoi, int *result);
  int convert_to_double(const char *const ctod, double *result);
};

// Provides an interface for creating akaroa Objects.
static class AkaroaClass : public TclClass {
public:
  AkaroaClass() : TclClass("Akaroa") {}
  TclObject* create(int, const char*const*) {
    return (new Akaroa());
  }
} class_akaroa;
```

```
// Interface to the random number generator of Akaroa
class AkRNGImplementation : public RNGImplementation
{
public:
  AkRNGImplementation(long s) {seed_ = s;};

  long next() // compatibility method
  {
    seed_ = (long) AkRandom();
    return seed_;
  };

  double next_double()   // Preferred method
  {
    return AkRandomReal();
  };

};

Akaroa::Akaroa()
{
  RNG::setRNGImplementation(new AkRNGImplementation(1L));
}

// The Tcl - C++ binding is done by this function
//
// Intention: Map all calls to the Tcl-Akaroa Object to
//   Akaroa library functions. See the documentation of
//   Akaroa for details.
//
// Input: parameters passed by ns. See ns Documentation for details.
//
// Output: Mostly none. Except when conversion fails.
//
// When calls to Akaroa fail, Akaroa will abort the simulation,
// providing its own diagnostics.

int Akaroa::command(int argc, const char*const* argv)
{

  int parameter = 0;
  double observation = 0;

  // Some function calls to akaroa take zero, one or two arguments.
```

```
switch (argc)
  {
  case 4:          // Evaluate function calls with two arguments first
      if (strcmp(argv[1], "AkObservation") == 0)
      {
        if(convert_to_int(argv[2], &parameter) == TCL_ERROR)
            return TCL_ERROR;

        if(convert_to_double(argv[3], &observation) == TCL_ERROR)
            return TCL_ERROR;

        // Tell AkObservation the parameter observed and the value
        AkObservation(parameter, observation);
        return TCL_OK;
      }
    break;

  case 3:        // one argument (less likely to occur)
    if (strcmp(argv[1], "AkObservation") == 0)
      {
        if(convert_to_double(argv[2], &observation) == TCL_ERROR)
            return TCL_ERROR;

        // Tell Akaroa about a new Observation
        // here we observe only one parameter
        AkObservation(observation);
        return TCL_OK;
      };

    if (strcmp(argv[1], "AkDeclareParameters") == 0)
      {
        if(convert_to_int(argv[2], &parameter) == TCL_ERROR)
            return TCL_ERROR;

        // Declare number of Parameters to be observed
        AkDeclareParameters(parameter);
        return TCL_OK;
      };

    if (strcmp(argv[1], "AkObservationType") == 0)
      {
        // Switch to a different Observation Type
        if (strcmp(argv[2], "AkIndependent") == 0)
          {
                AkObservationType(AkIndependent);
```

```
                return TCL_OK;
            }
          else if (strcmp(argv[2], "AkCorrelated") == 0)
            {
                AkObservationType(AkCorrelated);
                return TCL_OK;
            }
          else
            {
              Tcl& tcl = Tcl::instance();
              tcl.resultf("Akaroa: Invalid AkObservation type: %s",argv[2]);
              return TCL_ERROR;
            }
        }
      break;
    }
  return (TclObject::command(argc, argv));
}

int Akaroa::convert_to_int(const char *const ctoi, int *result)
{
  char *endptr = NULL;
  errno = 0;
  *result = strtol(ctoi,&endptr,0);
  if(*endptr == int(ctoi))
    {
      Tcl& tcl = Tcl::instance();
      tcl.resultf("Akaroa: Problems converting %s to integer (invalid string)",
                  ctoi);
      return TCL_ERROR;
    }
  if(errno)
    {
      Tcl& tcl = Tcl::instance();
      tcl.resultf("Akaroa: Problems converting %s to integer (out of range).",
                  ctoi);
      return TCL_ERROR;
    }
  return TCL_OK;
}

int Akaroa::convert_to_double(const char *const ctod, double *result)
{
  char *endptr = NULL;
  errno = 0;
```

```
  *result = strtod(ctod,&endptr);
  if(*endptr == int(ctod))
    {
      Tcl& tcl = Tcl::instance();
      tcl.resultf("Akaroa: Problems converting %s to double (invalid string)",
                  ctod);
      return TCL_ERROR;
    }
  if(errno)
    {
      Tcl& tcl = Tcl::instance();
      tcl.resultf("Akaroa: Problems converting %s to double (out of range)",
                  ctod);
      return TCL_ERROR;
    }
  return TCL_OK;
}
```

.2 rng.h

```
/* -*-  Mode:C++; c-basic-offset:8; tab-width:8; indent-tabs-mode:t -*- */
/*
 * Copyright (c) 1997 Regents of the University of California.
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. All advertising materials mentioning features or use of this software
 *    must display the following acknowledgement:
 *      This product includes software developed by the Computer Systems
 *      Engineering Group at Lawrence Berkeley Laboratory.
 * 4. Neither the name of the University nor of the Laboratory may be used
 *    to endorse or promote products derived from this software without
 *    specific prior written permission.
```

```
 *
 * THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ``AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 *
 * "@(#) $Header: /nfs/jade/vint/CVSROOT/ns-2/rng.h,v 1.18 2000/09/15 20:46:12 haoboy Exp $
 */

/* new random number generator */


#ifndef _rng_h_
#define _rng_h_


// Define rng_test to build the test harness.
#define rng_test

#include <math.h>
#include <stdlib.h>                     // for atoi

#ifndef stand_alone
#include "config.h"
#endif   /* stand_alone */

/*
 * RNGImplementation is internal---do not use it, use RNG.
 *
 * Prepared for inheritance Andreas Koepke, 2001-07-04,
 * declared next and next_double virtual and seed_ protected
 */
class RNGImplementation {
public:
        RNGImplementation(long seed = 1L) {
                seed_ = seed;
        };
        void set_seed(long seed) { seed_ = seed; }
```

```
        long seed() { return seed_; }
        virtual long next();                    // return the next one
        virtual double next_double();
protected:
        long seed_;
};

/*
 * Use class RNG in real programs.
 *
 * Changed stream_ from RNGImplementation to
 * RNGImplementation *, made it static.
 * Added reference counting. akoepke
 */
class RNG
#ifndef stand_alone
: public TclObject
#endif  /* stand_alone */
        {

public:
        enum RNGSources { RAW_SEED_SOURCE, PREDEF_SEED_SOURCE, HEURISTIC_SEED_SOURCE };

        RNG(); // : stream_(1L) {}; stream_ now pointer.
        RNG(RNGSources source, int seed = 1) { set_seed(source, seed); };

        void set_seed(RNGSources source, int seed = 1);
        inline int seed() { return stream_->seed(); }
        inline static RNG* defaultrng() { return (default_); }

#ifndef stand_alone
        int command(int argc, const char*const* argv);
#endif  /* stand_alone */
        // These are primitive but maybe useful.
        inline int uniform_positive_int() {  // range [0, MAXINT]
                return (int)(stream_->next());
        }
        inline double uniform_double() { // range [0.0, 1.0)
                return stream_->next_double();
        };

        // Added 2000-12-15, akoepke
        static void setRNGImplementation(RNGImplementation *imp);
        ~RNG();  // RNG owns a dynamic object
```

```
        // these are for backwards compatibility
        // don't use them in new code
        inline int random() { return uniform_positive_int(); }
        inline double uniform() {return uniform_double();}

        // these are probably what you want to use
        inline int uniform(int k)
                { return (uniform_positive_int() % (unsigned)k); }
        inline double uniform(double r)
                { return (r * uniform());}
        inline double uniform(double a, double b)
                { return (a + uniform(b - a)); }
        inline double exponential()
                { return (-log(uniform())); }
        inline double exponential(double r)
                { return (r * exponential());}
        inline double pareto(double scale, double shape)
                { return (scale * (1.0/pow(uniform(), 1.0/shape)));}
        inline double paretoII(double scale, double shape)
                { return (scale * ((1.0/pow(uniform(), 1.0/shape)) - 1));}
        double normal(double avg, double std);
        inline double lognormal(double avg, double std)
                { return (exp(normal(avg, std))); }

protected: // need to be public?

        static RNGImplementation *stream_;
        static RNG* default_;
        static int refCount;  // needed to clean up the stream_
};
/*
 * Create an instance of this class to test RNGImplementation.
 * Do .verbose() for even more.
 */
#ifdef rng_test
class RNGTest {
public:
        RNGTest();
        void verbose();
        void first_n(RNG::RNGSources source, long seed, int n);
};
#endif /* rng_test */

#endif /* _rng_h_ */
```

.3 rng.cc

```
/* -*-  Mode:C++; c-basic-offset:8; tab-width:8; indent-tabs-mode:t -*- */
/*
 * Copyright (c) 1997 Regents of the University of California.
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. All advertising materials mentioning features or use of this software
 *    must display the following acknowledgement:
 *      This product includes software developed by the Computer Systems
 *      Engineering Group at Lawrence Berkeley Laboratory.
 * 4. Neither the name of the University nor of the Laboratory may be used
 *    to endorse or promote products derived from this software without
 *    specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ``AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 */

#ifndef lint
static const char rcsid[] =
    "@(#) $Header: /nfs/jade/vint/CVSROOT/ns-2/rng.cc,v 1.20 2000/07/21 04:56:58 yewei Exp $
#endif
```

```
/* new random number generator */

#ifndef WIN32
#include <sys/time.h>                    // for gettimeofday
#include <unistd.h>
#endif

#include <stdio.h>
#include "rng.h"
#include "config.h"  // for gettimeofday

#ifndef MAXINT
#define MAXINT  2147483647      // XX [for now]
#endif


/*
 * RNGImplementation
 */

/*
 * Generate a periodic sequence of pseudo-random numbers with
 * a period of 2^31 - 2.  The generator is the "minimal standard"
 * multiplicative linear congruential generator of Park, S.K. and
 * Miller, K.W., "Random Number Generators: Good Ones are Hard to Find,"
 * CACM 31:10, Oct. 88, pp. 1192-1201.
 *
 * The algorithm implemented is:  Sn = (a*s) mod m.
 *   The modulus m can be approximately factored as: m = a*q + r,
 *   where q = m div a and r = m mod a.
 *
 * Then Sn = g(s) + m*d(s)
 *   where g(s) = a(s mod q) - r(s div q)
 *   and d(s) = (s div q) - ((a*s) div m)
 *
 * Observations:
 *   - d(s) is either 0 or 1.
 *   - both terms of g(s) are in 0, 1, 2, . . ., m - 1.
 *   - |g(s)| <= m - 1.
 *   - if g(s) > 0, d(s) = 0, else d(s) = 1.
 *   - s mod q = s - k*q, where k = s div q.
 *
 * Thus Sn = a(s - k*q) - r*k,
 *   if (Sn <= 0), then Sn += m.
```

```
 *
 * To test an implementation for A = 16807, M = 2^31-1, you should
 *   get the following sequences for the given starting seeds:
 *
 *   s0, s1,    s2,        s3,          . . . , s10000,     . . . , s551246
 *    1, 16807, 282475249, 1622650073,  . . . , 1043618065, . . . , 1003
 *    1973272912, 1207871363, 531082850, 967423018
 *
 * It is important to check for s10000 and s551246 with s0=1, to guard
 * against overflow.
*/
/*
 * The sparc assembly code [no longer here] is based on Carta, D.G., "Two Fast
 * Implementations of the 'Minimal Standard' Random Number
 * Generator," CACM 33:1, Jan. 90, pp. 87-88.
 *
 * ASSUME that "the product of two [signed 32-bit] integers (a, sn)
 *       will occupy two [32-bit] registers (p, q)."
 * Thus: a*s = (2^31)p + q
 *
 * From the observation that: x = y mod z is but
 *   x = z * the fraction part of (y/z),
 * Let: sn = m * Frac(as/m)
 *
 * For m = 2^31 - 1,
 *   sn = (2^31 - 1) * Frac[as/(2^31 -1)]
 *      = (2^31 - 1) * Frac[as(2^-31 + 2^-2(31) + 2^-3(31) + . . .)]
 *      = (2^31 - 1) * Frac{[(2^31)p + q] [2^-31 + 2^-2(31) + 2^-3(31) + . .
.]}
 *      = (2^31 - 1) * Frac[p+(p+q)2^-31+(p+q)2^-2(31)+(p+q)3^(-31)+ . . .]
 *
 * if p+q < 2^31:
 *   sn = (2^31 - 1) * Frac[p + a fraction + a fraction + a fraction + . . .]
 *      = (2^31 - 1) * [(p+q)2^-31 + (p+q)2^-2(31) + (p+q)3^(-31) + . . .]
 *      = p + q
 *
 * otherwise:
 *   sn = (2^31 - 1) * Frac[p + 1.frac . . .]
 *      = (2^31 - 1) * (-1 + 1.frac . . .)
 *      = (2^31 - 1) * [-1 + (p+q)2^-31 + (p+q)2^-2(31) + (p+q)3^(-31) + . . .]
 *      = p + q - 2^31 + 1
 */


#define A       16807L          /* multiplier, 7**5 */
```

```
#define M       2147483647L     /* modulus, 2**31 - 1; both used in random */
#define INVERSE_M ((double)4.656612875e-10)     /* (1.0/(double)M) */


long
RNGImplementation::next()
{
        long L, H;
        L = A * (seed_ & 0xffff);
        H = A * (seed_ >> 16);

        seed_ = ((H & 0x7fff) << 16) + L;
        seed_ -= 0x7fffffff;
        seed_ += H >> 15;

        if (seed_ <= 0) {
                seed_ += 0x7fffffff;
        }
        return(seed_);
}

double
RNGImplementation::next_double()
{
        long i = next();
        return i * INVERSE_M;
}

/*
 * RNG implements a nice front-end around RNGImplementation
 */
#ifndef stand_alone
static class RNGClass : public TclClass {
public:
        RNGClass() : TclClass("RNG") {}
        TclObject* create(int, const char*const*) {
                return(new RNG());
        }
} class_rng;
#endif /* stand_alone */

/* default RNG */

RNG* RNG::default_ = NULL;
RNGImplementation *RNG::stream_ = 0;
```

```
int RNG::refCount = 0;

RNG::RNG()
{
        if((refCount++ == 0) && (stream_ == 0))
                stream_ = new RNGImplementation(1L);
};

double
RNG::normal(double avg, double std)
{
        static int parity = 0;
        static double nextresult;
        double sam1, sam2, rad;

        if (std == 0) return avg;
        if (parity == 0) {
                sam1 = 2*uniform() - 1;
                sam2 = 2*uniform() - 1;
                while ((rad = sam1*sam1 + sam2*sam2) >= 1) {
                        sam1 = 2*uniform() - 1;
                        sam2 = 2*uniform() - 1;
                }
                rad = sqrt((-2*log(rad))/rad);
                nextresult = sam2 * rad;
                parity = 1;
                return (sam1 * rad * std + avg);
        }
        else {
                parity = 0;
                return (nextresult * std + avg);
        }
}

#ifndef stand_alone
int
RNG::command(int argc, const char*const* argv)
{
        Tcl& tcl = Tcl::instance();

        if (argc == 3) {
                if (strcmp(argv[1], "testint") == 0) {
                        int n = atoi(argv[2]);
                        tcl.resultf("%d", uniform(n));
                        return (TCL_OK);
```

```
                }
                if (strcmp(argv[1], "testdouble") == 0) {
                        double d = atof(argv[2]);
                        tcl.resultf("%6e", uniform(d));
                        return (TCL_OK);
                }
                if (strcmp(argv[1], "seed") == 0) {
                        int s = atoi(argv[2]);
                        // NEEDSWORK: should be a way to set seed to PRDEF_SEED_SORUCE
                        if (s) {
                                if (s <= 0 || (unsigned int)s >= MAXINT) {
                                        tcl.resultf("Setting random number seed to known bac
                                        return TCL_ERROR;
                                };
                                set_seed(RAW_SEED_SOURCE, s);
                        } else set_seed(HEURISTIC_SEED_SOURCE, 0);
                        return(TCL_OK);
                }
        } else if (argc == 2) {
                if (strcmp(argv[1], "next-random") == 0) {
                        tcl.resultf("%u", uniform_positive_int());
                        return(TCL_OK);
                }
                if (strcmp(argv[1], "seed") == 0) {
                        tcl.resultf("%u", stream_->seed());
                        return(TCL_OK);
                }
                if (strcmp(argv[1], "default") == 0) {
                        default_ = this;
                        return(TCL_OK);
                }
#if 0
                if (strcmp(argv[1], "test") == 0) {
                        if (test())
                                tcl.resultf("RNG test failed");
                        else
                                tcl.resultf("RNG test passed");
                        return(TCL_OK);
                }
#endif
        } else if (argc == 4) {
                if (strcmp(argv[1], "seed") == 0) {
                        int s = atoi(argv[3]);
                        if (strcmp(argv[2], "raw") == 0) {
                                set_seed(RNG::RAW_SEED_SOURCE, s);
```

```
                        } else if (strcmp(argv[2], "predef") == 0) {
                                set_seed(RNG::PREDEF_SEED_SOURCE, s);
                                // s is the index in predefined seed array
                                // 0 <= s < 64
                        } else if (strcmp(argv[2], "heuristic") == 0) {
                                set_seed(RNG::HEURISTIC_SEED_SOURCE, 0);
                        }
                        return(TCL_OK);
                }
                if (strcmp(argv[1], "normal") == 0) {
                        double avg = strtod(argv[2], NULL);
                        double std = strtod(argv[3], NULL);
                        tcl.resultf("%g", normal(avg, std));
                        return (TCL_OK);
                }
                if (strcmp(argv[1], "lognormal") == 0) {
                        double avg = strtod(argv[2], NULL);
                        double std = strtod(argv[3], NULL);
                        tcl.resultf("%g", lognormal(avg, std));
                        return (TCL_OK);
                }
        }
        return(TclObject::command(argc, argv));
}
#endif /* stand_alone */

void
RNG::set_seed(RNGSources source, int seed)
{
        /* The following predefined seeds are evenly spaced around
         * the 2^31 cycle.  Each is approximately 33,000,000 elements
         * apart.
         */
#define N_SEEDS_ 64
        static long predef_seeds[N_SEEDS_] = {
                1973272912L,  188312339L, 1072664641L,  694388766L,
                2009044369L,  934100682L, 1972392646L, 1936856304L,
                1598189534L, 1822174485L, 1871883252L,  558746720L,
                605846893L, 1384311643L, 2081634991L, 1644999263L,
                773370613L,  358485174L, 1996632795L, 1000004583L,
                1769370802L, 1895218768L,  186872697L, 1859168769L,
                349544396L, 1996610406L,  222735214L, 1334983095L,
                144443207L,  720236707L,  762772169L,  437720306L,
                939612284L,  425414105L, 1998078925L,  981631283L,
                1024155645L,  822780843L,  701857417L,  960703545L,
```

```
        2101442385L, 2125204119L, 2041095833L,   89865291L,
        898723423L, 1859531344L,  764283187L, 1349341884L,
        678622600L,  778794064L, 1319566104L, 1277478588L,
        538474442L,  683102175L,  999157082L,  985046914L,
        722594620L, 1695858027L, 1700738670L, 1995749838L,
        1147024708L,  346983590L,  565528207L,  513791680L
};
static long heuristic_sequence = 0;

switch (source) {
case RAW_SEED_SOURCE:
        if (seed <= 0 || (unsigned int)seed >= MAXINT)    // Wei Ye
                abort();
        // use it as it is
        break;
case PREDEF_SEED_SOURCE:
        if (seed < 0 || seed >= N_SEEDS_)
                abort();
        seed = predef_seeds[seed];
        break;
case HEURISTIC_SEED_SOURCE:
        timeval tv;
        gettimeofday(&tv, 0);
        heuristic_sequence++;   // Always make sure we're different than last time.
        seed = (tv.tv_sec ^ tv.tv_usec ^ (heuristic_sequence << 8)) & 0x7fffffff;
        break;
};
// set it
// NEEDSWORK: should we throw out known bad seeds?
// (are there any?)
if (seed < 0)
        seed = -seed;
stream_->set_seed(seed);

// Toss away the first few values of heuristic seed.
// In practice this makes sequential heuristic seeds
// generate different first values.
//
// How many values to throw away should be the subject
// of careful analysis.  Until then, I just throw away
// "a bunch".  --johnh
if (source == HEURISTIC_SEED_SOURCE) {
        int i;
        for (i = 0; i < 128; i++) {
                stream_->next();
```

```
                };
        };
}


/*
 * RNGTest:
 * Make sure the RNG makes known values.
 * Optionally, print out some stuff.
 *
 * Simple test program:
 * #include "rng.h"
 * void main() { RNGTest test; test.verbose(); }
 */

#ifdef rng_test
RNGTest::RNGTest()
{
        RNG rng(RNG::RAW_SEED_SOURCE, 1L);
        int i;
        long r;

        for (i = 0; i < 10000; i++)
                r = rng.uniform_positive_int();

        if (r != 1043618065L)
                abort();

        for (i = 10000; i < 551246; i++)
                r = rng.uniform_positive_int();

        if (r != 1003L)
                abort();
}

void
RNGTest::first_n(RNG::RNGSources source, long seed, int n)
{
        RNG rng(source, seed);

        for (int i = 0; i < n; i++) {
                int r = rng.uniform_positive_int();
                printf("%10d ", r);
        };
        printf("\n");
```

```
}

void
RNGTest::verbose()
{
        printf ("default: ");
        first_n(RNG::RAW_SEED_SOURCE, 1L, 5);

        int i;
        for (i = 0; i < 64; i++) {
                printf ("predef source %2d: ", i);
                first_n(RNG::PREDEF_SEED_SOURCE, i, 5);
        };

        printf("heuristic seeds should be different from each other and on each run.\n");
        for (i = 0; i < 64; i++) {
                printf ("heuristic source %2d: ", i);
                first_n(RNG::HEURISTIC_SEED_SOURCE, i, 5);
        };
}
#endif /* rng_test */

// Running simulations in parallel requires central
// place to deal with random numbers.
// This is done best using a central random number stream.
// So here is the function to replace the internal stream.
// akoepke 2000-12-15
void
RNG::setRNGImplementation(RNGImplementation* imp)
{
        if((imp != 0) && (imp != stream_))
        {
                if(stream_) delete stream_;
                stream_ = imp;
        }
        return;
};

RNG::~RNG()
{
        if(--refCount == 0)
        {
                if(stream_ != 0) delete stream_;
                stream_ = 0;
        }
```

```
}
```