Published at the 2nd Intl. OMNeT++ Workshop.
January 8-9, 2002
Technical University Berlin
Berlin, Germany

Using Akaroa2 with OMNeT++

Steffen Sroka and Holger Karl

Telecommunication Networks Group
Technical University Berlin
Berlin, Germany
sroka@ft.tu-berlin.de, karl@ee.tu-berlin.de
http://www-tkn.ee.tu-berlin.de

1 Introduction

When using discrete event simulation (DES) two main problems are: When is it ready to start collecting data (in order not to include initialization effects) and when to terminate the simulation? One possible criterion is given by the confidence level, more precisely, by its width relative to the mean. But ex ante it is unknown how many observations have to be collected to achieve this level. Another problem is that DES can consume much time. Even with today's high speed processors simulation of modest complexity can take hours. Akaroa2 [1] is a software packet that solves both problems. Akaroa was designed at the University of Canterbury in Christchurch, New Zealand and can be used free of charge for teaching and non-profit research activities. It is designed for running quantitative stochastic discrete event simulations on Unix multiprocessor systems or networks of heterogeneous Unix workstations. To speed up sequential simulation it uses *multiple replications in parallel (MRIP)*. This means that multiple instances of a sequential simulation program run on different processors. These instances run independently of one another and continuously send their observations to a central management process. This management process calculates from these observations an overall estimate of the mean value of each parameter. Akaroa2 decides by a given confidence level and precision whether it has enough observations or not. When it judges that is has enough observations it halts the simulation. The simulation would be sped up approximately in proportion to the number of processors used and sometimes even more.

Currently these functionality is not present within OMNeT++ [2] and implementing these algorithms by hand is complicated and error-prone. Therefore it seems to be a good idea to integrate the Akaroa2 capabilities into OMNeT++. Sequential simulation programs to be run under Akaroa must be written in either C or C++, or be capable of calling library routines written in C++. Therefore OMNeT++ simulations are good candidates to use with Akaroa.

This document shows that OMNeT++ simulations can benefit from the capabilities of Akaroa2 and describes a method to integrate the Akaroa2 functions into existing simulations. Section 2 briefly describes how Akaroa works. Afterwards I describe a simple class that integrates the Akaroa functionality into OMNeT++.

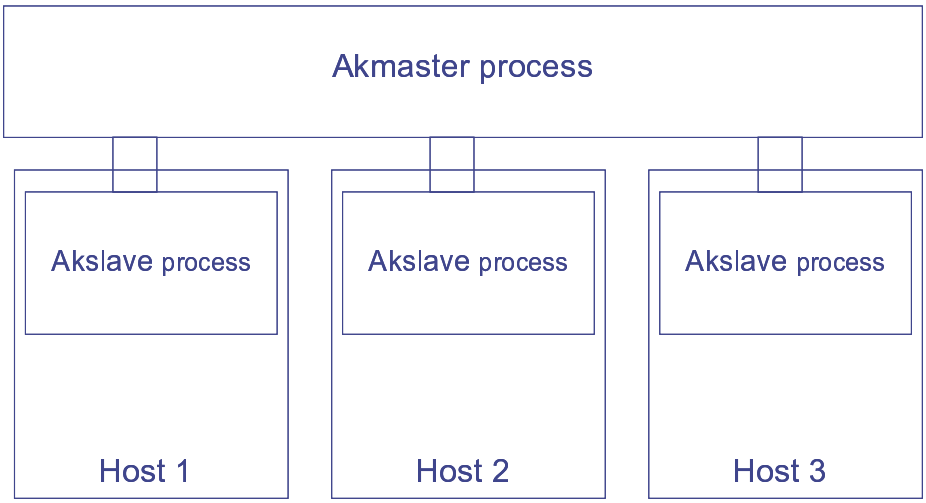

Fig. 1. Akaroa process structure

2 Running a simulation under Akaroa

This section is based on the manual of Akaroa [3] and the figures are borrowed form [4].

2.1 Parts of the Akaroa system

- Akmaster is the master process which coordinates all other processes in the Akaroa system.
- Akslave is a process that must run on each host that should be used for the simulation.
- Akrun instructs akmaster to launch the simulation on the requested number of hosts.

2.2 Starting up the Akaroa system

Before a simulation can be run in parallel under Akaroa, it is necessary to start the system up:

1. Start akmaster running in the background on some host.
2. On each host where you want to run a simulation engine, start akslave in the background.

Each akslave establishes a connection with the akmaster. Figure 1 shows the situation after starting up the system.

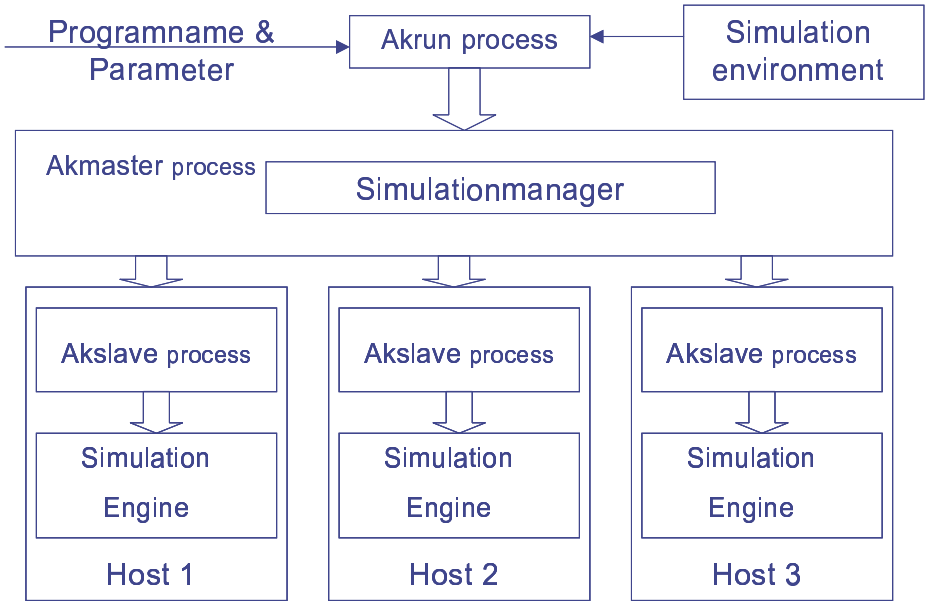

Fig. 2. Launching simulation engines

2.3 Running a simulation

The akrun command starts a simulation, waits for it to complete, and writes a report of the results to the standard output. The basic usage of the akrun command is:

```
akrun -n num_hosts command [argument..]
```

where `command` is the name of the simulation you want to start. If the simulation is not in the search path you should use the full name to invoke. When akrun starts it reads the file `Akaroa` in the working directory. There are some variables that customize Akaroa. For more details see the Akaroa manual.

The akmaster creates a simulation manager to coordinate activity related to this simulation. The simulation manager chooses the requested number of akslave processes and tells each one to launch an instance of the simulation program. Figure 2 illustrates what happens when invoking the akrun command.

The simulation manager monitors the global estimates of all the parameters, and when they have all reached the required precision, it tells all the simulation engines to terminate, breaks their connections, and sends the final global estimates back to the akrun process. Akrun then writes a report to standard output and exits. This final step is shown in Figure 3.

Here is a little example which shows the output when starting the simulation uni on two hosts [3].

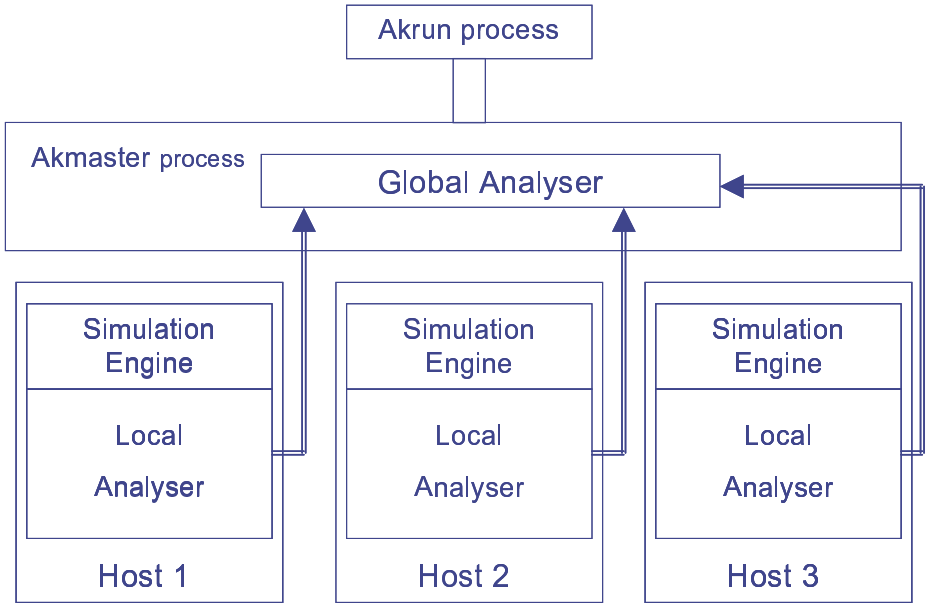

Fig. 3. Diagram of Akaroa running on 3 hosts

```
 whio% akrun -n 2 uni
Simulation ID = 17 Simulation engine started: host = pukeko, pid =
23672 Simulation engine started: host = purau, pid = 434
Param    Estimate       Delta  Conf         Var   Count   Trans
    1    0.503476   0.0157353  0.95 4.42582e-05    1530     255
whio%
```

2.4 Shutting down the Akaroa system

For shutting down the Akaroa system it is only necessary to kill the akmaster process. Any other processes (akslave, akrun or simulation engines) attached to the akmaster process will be automatically terminated.

2.5 Additional parts of Akaroa

- Akadd adds machines to a running simulation.
- Akstat provides information about a running simulation.
- Akgui is a GUI written in Python, but actually does not work with some Linux installation.

3 Writing an OMNeT++ simulation for Akaroa

One basic virtue of Akaroa is that it is easy to adapt existing simulation programs to run under it. Only three steps have to be taken:

– Because Akaroa use MRIP it is indispensable that all simulation runs independently. When using the build-in RNGs from OMNeT++ every replication would work with the same stream of random numbers. Consequently the simulation should always obtain random numbers from the Akaroa system. It uses a Combined Multiple Recursive pseudorandom number generator (CMRG) with a period of approximately 2^{191} random numbers and provides a unique stream of random numbers for every simulation engine.
– The Akaroa system needs to know how many parameters are to observe. This information has to be transfered before the first observation is made. The function `AkDeclareParameters(n)` tells Akaroa that is has to handle `n` parameters.
– Finally all observations have to be transmitted to Akaroa. This is solved by the function `AkObservation(i,x)` that collects observations for parameter `i`.

3.1 Sub-classing AkOutVector from cOutVector

For an easy use of the Akaroa functionality a sub-classing from cOutVector seems to be ideal because this is the OMNeT++ class where such continuous observations are recorded. Thereby the last two steps from above can be hidden from the user. Then only replacing cOutVector by AkOutVector and the replacing of the random generators are required. Through sub-classing all functionality from cOutVector remains available. If only a single replication of a program is used, the replacing of RNGs is not required.

Listing 1 shows the declaration of AkOutVector. When calling the constructor of AkOutVector the constructor of cOutVector is called too. Thus all functionality of cOutVector is achieved. In order to tell Akaroa how many parameters have to be observed `veccount` is a static member. When the simulation records its first observation AkOutVector calls `AkDeclareParameters(veccount)` to tell Akaroa the number of parameters. Every time the simulation calls `record()` the class transfers the value to Akaroa and OMNeT++.

When running different instances of one simulation working on the same directory the file names for storing data have to be different. Therefore the function `setOutFilename()` sets the file names for storing scalar and vector output `<hostname>.vec` and `<hostname>.sca`.

3.2 Random Numbers

When running multiple replications of a simulation model in parallel, it is important that each simulation engine uses a unique stream of random numbers, independent of the streams used by other simulation engines. For this reason, if your simulation requires random numbers, you should *always* obtain them from the Akaroa system, so that Akaroa can coordinate the random number streams received by different simulation engines. The Akaroa library provides several distributions (see Listing 2).

Listing 1. Declaration of AkOutVector

```
class AkOutVector : public cOutVector  {
   static bool Ak_declared;
   //is true when the number of records is already declared to Akaroa
   static long veccount;
   //number of parameters to be declared to Akaroa
   static bool hostfile;
   //is true when setOutFilename() has been called
   long Ak_id;
   //id of this parameter used by Akaroa
   void setOutFilename();
   //set the file name for *.vec and *.sca to <$HOST>.*
public:
        AkOutVector(const char *name=NULL) ;
        ~AkOutVector();
        virtual void record(double value);
};
```

Listing 2. Random distributions

```
#include <akaroa/distribution.H>

real Uniform(real a, real b);
long UniformInt(long n0, long n1);
long Binomial(long n, real p);
real Exponential(real m);
real Erlang(real m, real s);
real HyperExponential(real m, real s);
real Normal(real m, real s);
real LogNormal(real m, real s);
long Geometric(real m);
real HyperGeometric(real m, real s);
long Poisson(real m);
real Weibull(real alpha, real beta);
```

3.3 Other Problems

If you use several instances of simple modules to form one estimate, you have to make AkOutVector* a static member of this simple modules class. This is a bit complicated because the constructor of cOutVector requires an other object. Listing 3 shows a possible solution.

Listing 3. AkOutVector as static member

```
class ak_module : public cSimpleModule
{
        Module_Class_Members (ak_module, cSimpleModule, 16384);
public:
        virtual void activity ();
        virtual void finish();
        virtual void initialize();
        static AkOutVector* pParameter;
};
Define_Module( ak_module );

AkOutVector* ak_module::pParameter=NULL;

void ak_module::initialize()
{
  if (pParameter==NULL) pParameter=new AkOutVector("pParameter");

}
```

3.4 Calling finish()

Akaroa simply terminates the simulation engines when it has enough observations. Therefore it sends a SIGQUIT. In the Akaroa packet provided by TU-Berlin the signal is changed to SIGINT. This gives OMNeT++ the opportunity to call finish(). Currently OMNeT++ does not call `finish()` when it receives SIGINT. To archive this a modification of cmdenv is required. A modified `cmdenv.cc, cmdenv.h` is included in the packet.

4 Known Problems and Future Work

A problem occurs when recording data within dynamically generated modules because the parameter has to be declared in advance to the Akaroa system. A

solution would be a parameter placed in `omnet.ini` that contains the number of parameters to be observed.

5 Installing Akaroa under Linux

The installation of Akaroa under Linux needs some modifications of the Akaroa source files. A modified packet of Akaroa is available under `www-tkn.ee.tu-berlin.de/research/omnet-akaroa/`. For a detailed installation instruction read the read.me file in this folder.

References

1. K. Pawlikowski; D. McNickle; G. Ewing; R. Lee; J. Jeong. Project akaroa. Technical report, Department of Computer Science, University of Canterbury, Christchurch, New Zealand, `http://www.cosc.canterbury.ac.nz/research/RG/net_sim/simulation_group/a%karoa/download.html`, 2001.
2. Andreas Varga. Omnet++ eiscrete event simulation system. Technical report, Technical University of Budapest, `www.hit.bme.hu/phd/vargaa/omnetpp/`, 2001.
3. Greg Ewing; Krzysztof Pawlikowski; Donald McNickle. *Akaroa 2.6 User's Manual*. `www.cosc.canterbury.ac.nz/research/RG`, July 2000.
4. K. Pawlikowski; D. McNickle; G. Ewing; R. Lee; J. Jeong. Architecture of the akaroa system. Technical report, Department of Computer Science, University of Canterbury, Christchurch, New Zealand, `http://www.cosc.canterbury.ac.nz/research/RG/net_sim/simulation_group/a%karoa/architecture.html`, 2001.