The ROme OpTimistic Simulator

Alessandro Pellegrini
pellegrini@dis.uniroma1.it

Roberto Vitali
vitali@dis.uniroma1.it

Francesco Quaglia
quaglia@dis.uniroma1.it

Dipartimento di Informatica e Sistemistica
Sapienza, Università di Roma

ABSTRACT

In this paper we present the ROme OpTimistic Simulator (ROOT-Sim), an open source simulation platform developed using C technology and targeted at Linux Systems. It is a general-purpose simulation environment, based on the Parallel Discrete Event Simulation paradigm, exploiting the Time Warp protocol and Optimistic Synchronization techniques, which can be used on dedicated SMP machines, Clusters, or even in Time Sharing Desktop Grids. It transparently offers all the services needed to simulate complex/stateful models implemented in standard ANSI-C, providing a simple and compact set of APIs to interface the application-level software with the actual services needed by any Time Warp application, and offers communication between the nodes of the (distributed) simulation architecture via Message Passing. Here we present the programming model the user is expected to agree with in order to produce simulation software compatible with the platform, along with some short examples to show the potential and simplicity of the platform. Furthermore, we present all the services offered by ROOT-Sim, and finally we present some performance evaluation based on some real simulation models.

1. THE PROGRAMMING MODEL

Simulation Models can be implemented as software modules using the ANSI-C programming standard (see [4]). This means that the user can develop an application as if it were completely sequential. It is the platform's duty to parallelize it using as many available resources as possible.

In particular, this entails the possibility to organize the code in as many function/files as needed, to perform any sort of I/O during the simulation (keeping in mind that I/O operations can degrade performance), to use dynamically allocated memory to build the simulation state, and so on and so forth. The only exceptions are that (i) the `volatile` qualifier becomes meaningless (i.e., there is no possibility for a variable to be modified outside the simulation platform, then a forced fetch would only entail an additional overhead), and (ii) internal `static` variables cannot be handled by the simulation platform (i.e., it is not possible to have a variable internal to a function survive after its termination). In addition, the user code is expected to rely only on application level code, therefore no *NIX system calls are expected to be found.

No entry points are required for the application-level code, i.e., no `main()` function must be implemented into the source code. In fact, entry points for the application code are specified by ad-hoc APIs, which will be discussed in the next section.

The application-level code must be compliant to the Time Warp protocol presented in [3]. This means that the simulation model must be partitioned into N Logical Processes (LP), uniquely identified by a number in the range $[0, N-1]$, and the global simulation state S must be partitioned into LP states S_i, such that $\bigcup_{i=0}^{N-1} S_i = S \wedge \bigcap_{i=0}^{N-1} S_i = \emptyset$. Concretely, this means that the application-level code must implement the actual logic on al LP, and its internal simulation state must be a-priori defined using a `struct`. No shared variables among LPs are allowed.

The actual simulation is based on the idea of "*event*": each LP process events, and its advance in the Logical Virtual Time (LVT) is connected to their exectution. Processes communicate via messages, which are sent to other ones (possibily on other nodes). ROOT-Sim, as the Time Warp protocol stands, provides for a logical identity between events and messages. This means that a message envelopes only events to be scheduled to other LPs.

Messages are fixed-size. In particular, the application-level code must provide a definition of a `struct` where the contents of a message (an event) must be specified. Furthremore, events must be of known types, i.e., each event must be associated with a unique ID number which must be specified whenever a message is sent.

The LPs' states can be made arbitrarily grown during the simulation just relying on standard `malloc`/`free` operations. This unique feature means that the user can produce standard code to design the simulation model, with the only requirement that every `malloc`'d memory region must be referred via a pointer in the `struct _lp_state_type` for the simulation platform to be able to correctly rollback previous simulation states. Respecting these rules, everything in the LPs' states will be rollbackable, excepting global variables. Although global variables are allowed to be used in the application-level code, in case of a rollback operation their value will not be restored. Still, if the user produces code

SIMUTOOLS 2010 Barcelona, Spain

which relies on global variables, the model will be anyway valid if it does not expect to find consistent-after-rollback values in them.

1.1 Application Level APIs

The set of APIs to allow communication between application-level code and the simulation kernel is very simple. It consist of one call function (namely, `ScheduleNewEvent()`), and two callback functions (namely, `ProcessEvent()` and `OnGVT()`), which must necessarily be used/implemented in the application. Those APIs have the following signature and semantics:

- `void ProcessEvent(int me, time_type now, int event_type, event_content_type *event_content, void *state)`:
  This callback supports the actual processing of simulation events, and is used by the kernel to give control to the application layer. `me` is the ID of the LP being scheduled, `now` is the current value for the local clock, `event_type` is the ID code for the event to be processed, `event_content` is the information regarding the event itself, and `state` is the current LP's state.

- `void ScheduleNewEvent(int receiver, int sender, time_type timestamp, int event_type, void *event_content, int event_size)`:
  This function allows injecting a new simulation event within the system, to be destined to whichever simulation object. `receiver` and `sender` denote the ID of the destination and source LPs, `timestamp` is the LVT associated with the event to be processed, `event_type`, `event_content`, and `event_size` allow the correct identification and delivery of the actual event.

- `int CheckTermination(lp_state_type *snapshot, int gid)`:
  This callback allows passing control to the application layer by also providing a committed snapshot of the simulation object. It implements a distributed termination control, where `snapshot` is a portion S_i of the global state S where a global predicate must be evaluated over a Committed and Consistent Global State (CCGS) according to [5].

In addition, ROOT-Sim provides two additional functions, `Random()` and `Expent()`, which can be used to generate random numbers according to a uniform and exponential distribution respectivley, following the Piece Wise Deterministic paradigm (see [1]), i.e. the same sequence of numbers will be deterministically produced if a rollback operation is performed, provided that the random seed is included in the LPs' state.

1.2 Code Examples

In this section we present some code snippets to implement a working ROOT-Sim application which models a set of N nodes connected as a mesh, sending packets randomly to each other. As mentioned, the first important thing is to define the possible events handled by the model, the content of an event message, and the structure of the state:

```
1  #include <ROOT-Sim.h>
2
3  #define INIT 0
4  #define PACKET 1
5
6  #define PACKETS 1000000
7
8  extern seed_type master_seed;
9
10 typedef struct _event_content_type {
11     time_type sent_at;
12 } event_content_type;
13
14 typedef struct _lp_state_type{
15     int pckt_count;
16     seed_type seed_state;
17 } lp_state_type;
```

Notice that in this application we allow just two events: `INIT`, sent by the simulation kernel to startup the simulation, and `PACKET`, which identifies the transit of a packet in the mesh. `PACKETS` is a macro that will be used in the termination check, while `master_seed` is an initial seed for random functions exposed by the platform.

Then, we must specify the actual logic for the `ProcessEvent()` callback. This is the only entry point at application level for processing events, so we must rely on a `switch` construct to demultiplex them:

```
18 void ProcessEvent(int me, time_type now, int event_type,
19                   event_content_type *event_content, void *ptr) {
20
21     event_content_type new_event_content;
22     lp_state_type *pointer = (lp_state_type*)ptr;
23     time_type timestamp;
24
25     switch(event_type) {
26         case INIT:
27             pointer = (lp_state_type *)malloc(sizeof(lp_state_type));
28             pointer->pckt_count = 0;
29             pointer->seed_state = master_seed;
30
31             timestamp = (time_type)(20*Random((unsigned long *) \
32                         &pointer->seed_state));
33             ScheduleNewEvent(me,me,timestamp,PACKET,NULL,0);
34
35             break;
36
37         case PACKET: {
38             pointer->pckt_count++;
39             new_event_content.sent_at = now;
40
41             int rcv = (N_PRC_TOT * Random((unsigned long *) \
42                       &pointer->seed_state ) );
43             timestamp = now + (Expent(((unsigned long *) \
44                         &pointer->seed_state), DELAY));
45             ScheduleNewEvent(rcv,me,timestamp,PACKET, \
46                       &new_event_content,sizeof(new_event_content));
47         }
48     }
49 }
```

The logic in the code is fairly simple: upon `INIT` event, the LP's state is `malloc`'d and initialized, and an initial packet is sent to the LP itself. Whenever a `PACKET` event is received, a local counter is increased, and a packet is sent back to a random LP in the simulation environment. Timestamps associated to these events are computed according to an exponential distribution, exploiting the internal `Expent()` function.

`CheckTermination()` is the second callback to be implemented in the application-level code, and it performs a local check on the LP's state. In particular, if the number of packets passed in the LP is smaller than `PACKETS`, it tells the simulation platform that the simulation cannot be halted yet:

```
50 int CheckTermination(lp_state_type *snapshot, int gid) {
```

```
51    if (snapshot->pckt_count < PACKETS)
52        return 0;
53    return 1;
54 }
```

2. OFFERED SERVICES

The simulation platform offers several facilities for supporting the model simulation in the most effective way. First of all, we want to emphasize how the user can rely on the standard malloc library to allocate dynamic memory within the simulation state. This strong transparency lets the users produce standard ANSI-C code, leaving the simulation platform the duty of managing this space-varying state[1].

High effectiveness in this approach is achieved via *hooking* techniques, i.e. all `malloc/free` calls are hooked and redirected to a wrapper. At the same time, the simulation platform is "*context-aware*", i.e., it has an internal state which distinguishes whether the current execution flow belongs to the application-level code or the platform's internals. In the former case, the call is redirected to the internal Memory Map Manager, which handles the allocation/deallocation operation in order to keep some additional information nedded to make it correctly rollbackable (see [8]). In the latter case, the call is directly sent to the underlying malloc library.

In addition, the Memory Map Manager is able to identify at runtime what memory areas (belonging to the simulation states) are modified. This is achieved via a fast compile/linking-time code instrumentation, which adds before each memory writing instruction (i.e., every `mov` with a memory address as a destination operand) a `call` to an internal assembly module which identifies the destination and the size of the memory region affected by the update[2] (see [6]). Through this feature, ROOT-Sim is able to provide the user with both incremental and full checkpointing capabilites, i.e. the possibility to save either the whole state of an LP, or just those modified (since the last log) areas. Both full and incremental logs can provide an efficient simulation behaviour, depending on the actual runtime dynamics. To better capture this aspect, ROOT-Sim offers an autonomic optimization subsystem, based either on an analytical formula (see [9]) targeted at Clusters and Desktop Grids, or on an evolutionary algorithm (see [7]) targeted at any system typology, even time-sharing Desktop Grids.

Third Party libraries are almost fully supported. The user is allowed to rely on any third party library, handled via parsing/hooking techniques at compile/linking time, given that they are implemented statelessly, i.e. they do not keep internal buffers and/or allocate memory during their call. Support for stateful third party libraries is currently under development and will be integrated in a future release.

ROOT-Sim offers the user the possibility to select the scheduling algorithm for event processing selection from a wide range of standard schedulers (e.g., round robin, smallest timestamp first, probabilistic scheduling, grain sensitive, ...), which can be selected depending on the actual simulation model in order to provide the best performance.

[1]This entails deallocating/reallocating memory in case of rollback operations.

[2]The instrumenting tool is suited at ELF object files format, targeting IA-32 and x86-64 architectures, thus covering most of the off-the-shelf machines around.

The simulation platform can be run as a stand-alone process, or as a deamon. In the latter case, it will dynamically load the simulation model (generated using a special-purpose compiler provided with the platform) and will propagate it to all the nodes belonging to the simulation network for execution. ROOT-Sim's daemon provides an interactive shell for runtime activation and selection of subsystems.

3. PERFORMANCE EVALUATION

In this section we provide an evaluation of the overall simualtion platform's performance. As a simulation model, we have adopted a parallel distributed *Game of Life* [2]. Each LP models a cell in the grid, and the simulation state keeps a variable which tells whether the cell is currently alive or not, and a linked list of entries which keeps track of the history of births and deaths, associated with the logical virtual time the event happend at. Initial configuration can be specified through a text file which is read upon simulation startup.

We have varied both the number of nodes in the simulation network and the grid size, in order to assess how the simulation platform scales, even when simulation states can grow arbitrarily.

4. REFERENCES

[1] M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. Technical report, ACM Computing Surveys, 1996.

[2] M. Gardner. Mathematical games: The fantastic combinations of John Conway's new solitaire game 'Life'. *Scientific American*, 223(4):120–123, 1970.

[3] D. R. Jefferson. Virtual Time. *ACM Transactions on Programming Languages and System*, 7(3):404–425, July 1985.

[4] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd edition, 1988.

[5] F. Mattern. Efficient algorithms for distributed snapshots and global virtual time approximation. *Journal of Parallel Distributed Computing*, 18(4):423–434, 1993.

[6] A. Pellegrini, R. Vitali, and F. Quaglia. Di-dymelor: Logging only dirty chunks for efficient management of dynamic memory based optimistic simulation objects. In *PADS '09: Proceedings of the 2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation*, pages 45–53, Washington, DC, USA, 2009. IEEE Computer Society.

[7] A. Pellegrini, R. Vitali, and F. Quaglia. An evolutionary algorithm to optimize log/restore operations within optimistic simulation platforms. In *SIMUTOOLS 2001, To Appear*, Barcelona, Spain, 2011. SIGSIM.

[8] R. Toccaceli and F. Quaglia. Dymelor: Dynamic memory logger and restorer library for optimistic simulation objects with generic memory layout. In *PADS '08: Proceedings of the 22nd Workshop on Principles of Advanced and Distributed Simulation*, pages 163–172, Washington, DC, USA, 2008. IEEE Computer Society.

[9] R. Vitali, A. Pellegrini, and F. Quaglia. Autonomic log/restore for advanced optimistic simulation systems. In *International Symposium on Modeling, Analysis, and Simulation of Computer Systems*, volume 0, pages 319–327, Los Alamitos, CA, USA, 2010. IEEE Computer Society.