NAG 2-593

Monitoring and Controlling Distributed Applications Using Lomita*
(Position Paper)

Keith Marzullo
Ida M. Szafranska

IN-61-CR
127086
P-8

TR 92-1306
October 1992

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

*This work was supported by the Defense Advanced Research Projects Agency (DoD) under NASA Ames grant number NAG 2-593, Contract N00140-87-C-8904, and by grants from IBM T.J. Watson, IBM Endicott, Xerox Webster Research Center and Siemens RTL. The views, opinions and findings contained in this report are those of the authors and should not be construed as an official Department of Defense position, policy or decision.

Monitoring and Controlling Distributed Applications using Lomita*

POSITION PAPER

Keith Marzullo Ida M. Szafranska

Cornell University
Department of Computer Science
Ithaca, NY 14853–7501

16 October 1992

Over the last four years, we have developed the Meta toolkit for controlling distributed applications. This toolkit has been publically available as part of the academic ISIS release, and has been used both within and outside of Cornell for building various system monitoring and control applications [5, 3, 4].

One major stumbling block with using Meta has been the language (called NPL) it supports. NPL is very low-level and using it is difficult, in the same way it is difficult to write machine language programs or raw Postscript programs. Hence, we have spent the last six months building a higher-level language and runtime environment. Our hope is that with this higher-level approach, we will be able to write more complicated Meta applications and thereby concentrate more on the use (and limitations) of Meta as an architecture.

This note proceeds as follows. In Section 1, we review the Meta toolkit and its intended use. In Section 2 we describe our goals with Lomita and give an overview of its architecture and language syntax. In Section 3 we give a detailed example of the use of Lomita by presenting a complete program for a load-adaptable service.

*This work was supported by the Defense Advanced Research Projects Agency (DoD) under NASA Ames grant number NAG 2-593, Contract N00140-87-C-8904, and by grants from IBM T.J. Watson, IBM Endicott, Xerox Webster Research Center, and Siemens RTL. The views, opinions, and findings contained in this report are those of the authors and should not be construed as an official Department of Defense position, policy, or decision.

1 Review of Meta

A reactive system architecture partitions the system into two components: an active *environment* and an input-driven *control program*. The control program monitors the state of the environment through a *sensor* abstraction, and when the state meets some condition then it alters the environment's state through an *actuator* abstraction. Process control systems naturally have a reactive architecture, as does system and network monitoring, software tool integration, debugging, and automatic system management.

The Meta toolkit assists in the construction of distributed and reliable (albeit non-real-time) reactive systems. With Meta, one can instrument a program with software sensors and actuators in order to expose its state for control. Then, a control program can be written to monitor and control the instrumented programs. The Meta architecture interprets the control program in a distributed manner in order to supply both lower latency and tolerance to partial failures of the environment. Furthermore, the monitoring and control is done in a way to guarantee that the observed global state is consistent and changed atomically with respect to the monitoring of the control program.

For example, consider a simple computation server that accepts jobs and executes them in

the order received (the pending job requests are kept in a queue). The load of a server is the estimated time needed to complete all submitted jobs. As well as being submitted, a job can be cancelled and the server can be stopped (losing all submitted jobs).

This server can be instrumented with a sensor that gives the load of the server and a sensor that gives the queue of submitted jobs. It can also be instrumented with two actuators: one that cancels a job and one that stops the server. Then, Meta can be used to construct a service out of servers—for example, an actuator can be defined that submits a job to the lightest-loaded server of a group of servers, and a sensor can be defined that gives the average load of a set of servers. And, a control program can be written that creates additional servers on lightly-loaded machines when the average load is too high. Section 3 develops this example more fully.

There are two steps to managing a distributed application with Meta: instrumenting the application and writing the control program. Instrumentation is the more straightforward task. A Meta sensor or actuator is simply a procedure that is added to the application, where a sensor has no side-effects and an actuator changes the state of the application and returns **success** or **failure**. These procedures are registered with Meta using a library routine, which also associates a name and a type signature with the sensor or actuator. Finally, Meta has a set of library routines that synchronizes the sampling of sensors and invocation of actuators with its own operation in order to guarantee that Meta sees and alters on only locally consistent states.

An instrumented program is an example of a Meta *context*, that is, a named set of sensors and actuators. In Meta, each context belongs to a single *context class* that defines the types of its sensors and actuators. For example, if we assume that only one computation server will be run on any given machine, then the load sensor of a computation server running on a machine grimnir could be named serv(grimnir).load, where the context is named serv(grimnir) and is of a context class named serv.

Instrumented programs define what is called in Meta *base contexts*. Base contexts can be grouped into *group contexts*[1]. Put another way, a group context class can be defined as a collection of contexts from the same base context class, and a base context can join and leave any number of group contexts of a compatible class. Each sensor of the base context class exists in the group context class except that the type of the sensor is promoted to a set value. For example, assume that *service(1)* is a group context class comprised of serv contexts. If the load sensor in the context class serv has an integer type, then service context class also has a load sensor but its type is set of integers. The value of this sensor in some group context is the set of load sensor values, one for each base context that is a member of the group context.

Similarly, the actuation of service(1).stop will actuate serv(x).stop for every serv(x) that is a member of service(1), and the value of the actuation is **success** if all base actuations succeed; else the value is **failure**. Actuators in group context classes can also take two additional parameters: a positive integer and a set of values obtained from a sensor of the group. The first parameter specifies a number k of base contexts and the second parameter specifies a preference ranking r of the base contexts indicated by the source of the individual value. The actuation will invoke the actuator on the first k contexts denoted by r. For each that returns **failure**, an additional context is chosen from r. The group actuation will return **success** if k base contexts return **success**. For example, service(1).shutdown(2, sort(load)) will shut down the two lightest-loaded servers that are members of service(1).

Control programs are written in a simple programming language called NPL. An NPL command is equivalent to an atomic guarded command $\langle \phi_1 \rightarrow \alpha_1 || \ldots || \phi_m \rightarrow \alpha_m \rangle$, where each ϕ_i is a predicate expression over sensor values and each α_i is a sequence of actuator invocations

[1] A group context is called an *aggregate* in Meta. Meta is somewhat confusing in terms of contexts and context classes, however, and so we use the (hopefully clearer) Lomita terminology here.

whose parameters can be expressions of sensor values. The meaning of such a command is that it blocks until some ϕ_i is true, at which point the corresponding α_i executes, and any effects of α_i are not visible to other guarded commands until α_i terminates. Such commands can be one-shot (once an α_i executes the command terminates) or iterative (once an α_i executes the command resumes waiting for a predicate to become true). Meta also guarantees that an NPL command observes a valid sequence of global states. That is, not only is each global state used to evaluate a ϕ_i a valid global state [1], but the sequence of states is also consistent with the actual run of the environment [2].

Each context has associated with it an interpreter of NPL commands. For base contexts, the interpreter resides in the same address space as the instrumented program. An NPL command can be run in any interpreter (that is, an NPL program using fully-qualified names can be submitted to any context without changing its meaning), although the latency due to network communication is large—a command may run up to 500 times slower in a remote context than in a local context. Of course, some programs refer to more than one context and so must refer to some remote sensor or actuator no matter in which context they are run.

Interpreters for group contexts are created by informing an interpreter that it should also implement the group context. For example, the interpreter for serv(grimnir) can be told to also implement the service(1) context. In addition, more than one interpreter can be so informed, in which case they run in a replicated mode—even though an interpreter fails, the context will remain accessible and the NPL commands it is running will continue to run.

2 Lomita

Although Meta is a powerful system, it is extremely awkward to use. The NPL programs one writes for even simple control programs are very hard to read and to validate their correctness. Our goal with Lomita is to provide enough syntax and supporting semantics in order to make Meta usable.

The central idea of Lomita is to fully implement the context class abstraction. Rather than submitting NPL programs to contexts, one writes a description of the context classes which includes a set of atomic commands (in a syntax much more readable than NPL). The Lomita runtime system then ensures that contexts are initialized and recovered with the appropriate NPL commands.

Lomita consists of two parts. First, there is a compiler that takes Lomita programs and produces an object file. Second, there is a replicated fault-tolerant service called the *Lomita runtime* that, when given a Lomita object file, loads the file into an internal database. The runtime monitors the currently active contexts and downloads the relevant NPL commands from its internal database when necessary. The runtime also creates interpreters for group contexts when they are needed.

A Lomita program consists of a set of context class definitions. Each context class definition specifies the attributes of the context class and lists the rules to be run in each context of that context class. Attributes can either be Meta sensors or actuators, they can be functions or they can be the Lomita *key* construct.

The example in Section 3 gives several context class definitions. For example, the definition of the *machine* context declares that there is an instrumented program that supplies sensors on the load of the machine and on who is logged in, and extends this context class with some additional sensors, such as when the machine is to be considered "busy". The definition also contains a single rule that initializes a value by invoking the "stop_server" actuator.

There are three different kinds of context classes that can be declared in a Lomita program: the global context class, base context classes, and group context classes. Each context class defines a set of attributes and rules that apply to all contexts of that class. Base context classes and group context classes correspond with their equivalent in Meta. The global

context class contains a single context, called the global context. The attributes defined in the global context are available in all contexts. For example, every context has its own *print* actuator, and so *print* is defined as an actuator of the global context.

Lomita rules has the following syntax:

if/when predicate expression
do sequence of actuator invocations **end**
[**else if/when** predicate expression
do sequence of actuator invocations **end**]*

By default, a Lomita rule is translated into an iterative guarded command, but a programmer can stop iteration by using the exit actuator. The difference between "if" and "when" corresponds to whether the action is enabled in any state satisfying the predicate expression or only in a state in which the predicate becomes true. For example, the Lomita rule

```
when "marzullo" in login
  do print("watch out!") end
```

prints the message "watch out!" once after each time "marzullo" logs in, while the rule

```
if "marzullo" in login
   do print("watch out!") end
```

continuously prints the message "watch out!" as long as "marzullo" is logged in.

Group context classes can also specify rules that are to be run in the base context of all members of a group. Such rules are specified by a **with** statement, which has the following syntax:

with expression/**all**
[**select when** predicate expression /
remove when predicate expression /
rule]* **end**

The expression following the **with** keyword is called the *key expression* and when evaluated in the base context, yields the value of the key associated with the group context. A **select** statement generates a rule for joining the group and **remove** generates a rule for leaving the group. For example, consider the following definition of a group of machines:

```
free_machines: machine group
 attributes
   key gp: string
   end
 with type
   select when ! busy
   remove when busy
   if timer(10000)
     do print(name,
       " has been free for 10 seconds.")
     end
end
```

The key for the group is the value of the *type* sensor, which yields the type of instrumented machine. Hence, this context class partitions machines into group contexts all containing the same type of machine. The rule in the **with** statement is run in each machine context that is a member of a free_machines context—in this case, a free machine will print every ten seconds that it is a free machine.

3 Example

The following is a complete Lomita 1.0 program. The program *serv* services a simple request for computation (the computation is given a name and an estimated amount of time). An instrumented server is a member of the context class *serv*, and the context is named by the machine it runs on (e.g., *serv(ydalir)*). Servers are grouped into two groups—the group of all servers, and the group of servers that are not overloaded (called *free_servers*). Furthermore, the new actuator *add1* defined in *free_servers* submits a job to the lightest-loaded free server.

A set of rules, associated with the group of all servers, governs the number of server replicas. These rules specify that the number of replicas must be between min_rep and max_rep. Furthermore, if the average load of the servers is too high, then a new server is created, and if the average load of the servers is too low and there is a server with no jobs, then that server is deleted.

```
#define high_load 5.0
#define max_users 2
```

```
#define dally 30
#define max_load 30
#define min_load 2
#define max_rep 5
#define min_rep 1
#define serv_cmd "/usr/meta/utils/serv &"
#define has_server get1
#define set_has_server set(1, TRUE)
#define set_no_server set(1, FALSE)

#define wait_new_size get1
#define set_new_size set(1, TRUE)
#define reset_new_size set(1, FALSE)
#define last_nservers get2
#define set_nservers set(2, num_servers)

global attributes
  sensor get1: boolean
  sensor get2: integer
  function avg (any): any
  function sort ({any}): {any}
  function timer (integer): boolean
  function select_eq_int  (
    {integer}, integer): {integer}
  actuator exit
  actuator set (integer, any)
  actuator print (any)
  actuator shell (any)
  end

machine: base
  attributes
    key name: string
    sensor load: real
    sensor alive: boolean
    sensor busy: boolean:= load > high_load
      || size(login) >  max_users
      || has_server
    sensor login: {string}
    actuator exec (cmd: string)
    actuator start_server:=
      exec(serv_cmd);
      set_has_server;
      leave("freemachines")
    actuator stop_server:= set_no_server
    end
  if true do stop_server; exit end
  end

serv: base
  attributes
    key name: string
    sensor load: integer
    sensor alive: boolean
    sensor queue: {string}
    sensor overload: boolean:=
      load > max_load
    actuator add (
      job_name: string, job_time: string)
    actuator remove (job_name: string)
    actuator shutdown
    actuator stop:= shutdown;
      machine(name).stop_server
    end
  end

/* all machines that aren't busy */
freemachines: machine group
  attributes
    key not_needed
    sensor mean_load: real:= avg(load)
    sensor num_freemachines:= size(alive)
    actuator start_server (
      number: integer, pref: any)
    end
    with all
      select when ! busy
      remove when busy
      if timer(dally*1000)
        do print(
          name,
          " has been free for ",
          dally, " seconds.") end
      end
  end

/* all servers that aren't overloaded */
/* actuator add1 submits job to lightest */
/*    loaded server. */
freeservers: serv group
  attributes
    key not_needed
    sensor num_freeservers:= size(alive)
    actuator add (
      number: integer, pref: {integer},
      job_name: string, job_time: string)
    actuator add1 (
      jname: string, jtime: string):=
      add (1, sort(load), jname, jtime);
    end
    with all
      select when !overload
      remove when overload
      end
  end
```

```
/* All servers. Create a server if the */
/* average load is too high, and destroy */
/* an idle server if the average load is */
/* too low. */
servers: serv group
  attributes
    key not_needed
    sensor num_servers:= size(alive)
    actuator add (
      number: integer, pref: {integer},
      job_name: string, job_time: string)
    actuator stop (
      number: integer, pref: {integer})
    end
    with all select all end

    if true do set_new_size; exit end
    when num_servers <> last_nservers
      do set_nservers; set_new_size end

    if wait_new_size
     && (freemachines.num_freemachines > 0)
     && (num_servers == BOTTOM
          || num_servers < min_rep
          || (avg(load) > max_load
             && num_servers < max_rep))
      do freemachines.start_server(
        1, sort(load));
        reset_new_size end

    if wait_new_size
      && (num_servers > max_rep
         || (avg(load) < min_load
             && num_servers > min_rep))
      do stop(1, select_eq_int(load, 0));
         reset_new_size end
  end
```

Acknowledgements Mark Wood was the co-designer and principle software architect of the original Meta system. Tim Clark designed and built the Lomita runtime system, and Sue Honig designed and built the interface between Lomita and Meta. Ken Birman, Robert Cooper and Fred B. Schneider have all contributed ideas to both Meta and Lomita.

References

[1] K. M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. ACM Transactions on Computer Systems, 3(1):63–75 (February 1985).

[2] K. Marzullo and G. Neiger. Detection of Global State Predicates. Proceedings of the Fifth Workshop on Distributed Algorithms and Graphs (Springer-Verlag LNCS 579) pp 254–272. Delphi, Greece, October 1991.

[3] K. Marzullo and M. Wood. Tools for Distributed Application Management. In Proceedings of the Spring 1991 EurOpen Conference, Tromso, Norway, May 1991, pp 185–196.

[4] K. Marzullo and M. Wood. Tools for Managing and Controlling Distributed Applications. Cornell University Department of Computer Science TR 91– 1187 (February 1991, submitted for journal publication).

[5] K. Marzullo, M. Wood, K. Birman and R. Cooper. Tools for Monitoring and Controlling Distributed Applications. IEEE Computer 24(8): 42–51 (August 1991).