Check for updates

Rule-based Management of Distributed Operating Systems

Mark D. Wood
wood@cs.vu.nl

Department of Mathematics and Computer Science
Vrije Universiteit
Amsterdam, The Netherlands

Abstract

This paper presents a paradigm for managing a distributed operating system using a rule-based architecture. Recent trends have led to the structuring of operating systems, particularly those for distributed systems, as a set of microkernels with much of the system functionality being provided by a set of servers operating in user space. The proliferation of client-server based systems can easily lead to a set of independent, non-cooperating servers, with no common technique for management. The operation of each server is often hard-coded into the server, with no facility for dynamic adaptation and management. A general-purpose, rule-based approach to server control fills the need for management and can even eliminate the need for some services.

1 Introduction

Distributed operating systems have in recent years been constructed using microkernel technology. Examples of operating systems designed around a microkernel include Amoeba [6, 9], Chorus [2] and Mach [7]. The microkernel contains only that functionality which must execute in the supervisory mode of the processor; the balance of the operating system's functions are provided by server processes. For example, the microkernel typically handles memory management and fine-grain task scheduling while special servers provide functions such as long-term job scheduling, file management, and system administration. Rather than receiving these services directly from the kernel, processes become clients of the appropriate server.

A microkernel-based distributed operating system typically requires many different servers. For example, the Amoeba system includes a run server for scheduling jobs, a file server, a directory server, and even a server for maintaining the servers, called the boot server. A distributed operating system must ensure that the appropriate servers are available; this is the function of the Amoeba `boot` server. To provide fault-tolerance or to improve system performance, a system may additionally wish to run multiple copies of some services.

Each server is usually responsible for the management of some resource, such as a disk or a set of CPU's. By dynamically adapting to current conditions, a server can improve its utilization of the resource. For example, the job scheduling server may perform process migration in order to balance the system load. Servers which support dynamic adaptations to current conditions usually do so in a way that is hard-coded into the server.

The microkernel design with its accompanying collection of servers can easily lead to a complicated set of system management problems; these problems will only become more acute as distributed systems continue to grow in size. For example, the Amoeba operating system shall shortly be running in a configuration containing well over a hundred pool processors. Ensuring that all the components of this system function properly will be a serious management task.

The problems of system management are not limited to architectures based upon the processor pool model of Amoeba. Distributed systems spanning a network of workstations likewise have a set of system-wide services requiring management. These systems will also

benefit from the approach argued for here.

2 Rule-based System Management

The problems of system management in a distributed operating system may be effectively solved by the use of a rule-based approach. This approach provides a uniform method for controlling diverse aspects of the system. Moreover, by utilizing a rule-based approach, the policies which control the system may be clearly separated from the mechanism that enforces those policies.

To implement a rule-based approach, the distributed operating system and its services are instrumented with routines that expose the state of the system. Borrowing terminology from the area of process control, we call these hooks into the system *sensors* and *actuators*, with sensors being routines that return some aspect of the system state and actuators being procedures that alter some aspect of the state. The exposed state is operated upon by a logical layer of control. Clearly any control system for a distributed operating system should itself be distributed and reliable.

The Meta toolkit [5] is one such system. This toolkit provides facilities for reliable management of general distributed applications [3]: a distributed operating system may be viewed as an example of a distributed application. Meta typifies the kind of management system that could be used to control a distributed operating system. We briefly discuss in the next section the main features of Meta.

3 The Meta System

The basic structure of a system being managed using Meta is as shown in Figure 1. Though the application layer and the control layer are shown as single boxes, each of these layers actually consists of multiple subcomponents, distributed through out the system.

A detailed discussion of the Meta architecture appears elsewhere [10], but we briefly review here the basic principles. Each program component of a distributed application is instrumented by linking in the Meta runtime library and adding a small amount of code to

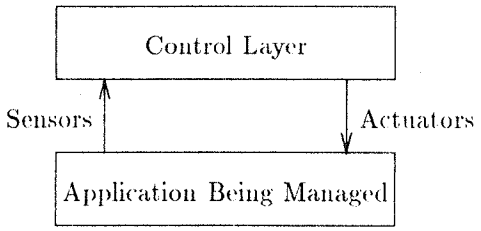

Figure 1: Structure of an application under Meta

the program. In particular, the programmer must add sensor and actuator routines which provide Meta with a uniform manner for accessing the system state. A number of predefined types (scalars, strings, sets, and intervals) are supported. Actuator routines return either success or failure. The Meta stub calls the sensor/actuator routines as necessary. The stub runs as a coroutine with the original program, requiring the programmer to add calls to the program which transfer control to the stub. This coroutine structure is typically not a problem for operating system services, as these servers usually have a main loop that listens for work. Instrumenting such a server with Meta only requires that a call to the Meta stub be added to the loop.

As an aside, we note that the Meta toolkit was originally developed for the Unix environment and included a facility based upon `ptrace` by which the state of an arbitrary program could be accessed without requiring recompilation. However, using a facility such as `ptrace` generally leads to inconsistencies and race conditions in the sampling of data, and so this approach is not recommended for most applications.

Each instrumented program component defines a logical entity for Meta, with a set of sensors and actuators. A set of different instances of the same program may be grouped together into an *aggregate*: an aggregate itself is treated by Meta as a single logical entity, inheriting the sensors and actuators of the program instances over which it is formed. A process may belong to multiple aggregates. For example, a job execution server for a specific

Sparc CPU might belong to the `FreeMachine` aggregate and also to the `Sparc` aggregate.

Control policies are expressed to the Meta system as a set of rules, written as guarded commands (see, *e.g.*, [1]). Each guarded command consists of a (*predicate*, *action*) pair, with the action taken when the global state of the system satisfies the predicate. Meta supports two types of guarded commands. In one, the action is enabled if the system state satisfies the predicate. In the other, the action is enabled only after the event of the predicate becoming true.

The language for guarded commands is a simple, postfix language, designed for ease of interpretation. Despite its simplicity, the language is quite expressive; arbitrary finite state automata may be expressed as a set of guarded commands.

Control policies are implemented by small interpreters running as part of each Meta stub. Rules governing the behavior of the distributed application may be assigned to any stub for execution, though for efficiency reasons, rules should be assigned in such a fashion as to minimize the number of remote references. The Meta library uses causality-preserving atomic broadcasts to provide globally-consistent handling of remote references. A weak notion of an atomic transaction is employed to carry out remote actuations.

In addition to stubs linked in with different system components, Meta supports free-standing stubs. Such stubs function as dedicated *Meta servers*. Aggregates are typically handled by Meta servers; guarded commands may also be assigned to them for execution. An important characteristic of Meta servers is that they may be run fault-tolerantly, by running multiple replicas. Any number of Meta servers may be run, and each server may be implemented by multiple replicas.

It is too expensive—and unnecessary—to have each stub continually be informed of the current global state of the application. Consequently, guarded commands are evaluated in Meta against *causally-consistent* views of the global state. (This is discussed in much more detail in [10].) Though stronger notions of global state detection can be defined [4], they are expected to be unnecessary for distributed operating system management. Likewise, Meta does not block a component from executing during the interval between when a condition is sensed and when the corresponding reaction actually takes place. A system which guarantees that global state detection and reaction are atomic can only do so with a high cost of blocking.

The relatively weak model used by Meta for detection and reaction was chosen to minimize the cost that the management system has on the application being controlled. Since the management system is being imposed on top of a preexisting application, it is crucial that the cost of monitoring and control be very low; otherwise the use of the management system becomes unacceptable.

Though the weak consistency guarantees provided by Meta are probably unacceptable for real-time process control, we expect that Meta's detection and reaction semantics will prove to be adequate for managing a distributed operating system. To justify this argument, we must consider in more detail the semantics of the system being managed. A distributed operating system has a number of characteristics which make the low-cost approach of Meta feasible.

- Many predicates of interest are *dependent monotonic* [8]. Such predicates, once true, will remain true until some external action is taken.

  For example, a manager for a distributed operating system might have rules detecting deadlock. Each deadlocked participant will be unable to make progress until a corrective action is taken. Consequently, if there exists a causally-consistent global state in which each participant is seen as being deadlocked, then the actual system is in fact deadlocked, and will remain so until the manager takes some corrective action.

  As another example, consider rules which restart failed services. The condition *all file servers have failed* is also dependent monotonic, remaining true until the manager restarts the servers.

- Many conditions within a distributed system change slowly. All that is required is

that the manager react in a "timely" fashion. In some cases, the observed state of the system is only used as a hint by the manager.

As an example, suppose the manager is to start a new task on the least loaded processor. It probably is not crucial that the manager know exactly what the load is on each processor, nor is it likely to be significant if another processor actually had a lower load at the instant the task was initiated.

Note that interaction between the manager and the system being managed is a feedback loop. The manager responds to conditions as it sees them, taking the appropriate actions, which results in the manager seeing a new state of the system. Errors in management can arise when the distributed control layer responds to conditions which are not dependent monotonic and which change quickly. Such errors may often be subsequently corrected. For example, suppose the manager is trying to run a job on the least loaded processor, but because of out-of-date information, incorrectly schedules the job on some other processor. Unless the scheduler's information is very dated, this error is likely to be of little consequence. However, if desired, such errors can be corrected; in this case, by migrating the process. Of course, corrective action is often expensive, and can lead to instabilities. Care must be taken in the specification of system rules to ensure that such problems do not arise. However, we believe that for most applications, the exercise of such care is preferable to using a management system which minimizes feedback problems by guaranteeing a stronger detection and reaction semantics.

4 Advantages of Rule-based Management

The rule-based approach argued for in this paper offers a number of benefits to the current ad-hoc approach to system management.

- It provides a uniform method for managing all aspects of the system.
- Policies may be dynamically updated without rebooting servers, or worse yet, the entire operating system. The system may even dynamically update its own rules. Moreover, the set of rules may be divided into different rule spaces, allowing individual users or subsystems to manage their own subspaces.
- The functionality of many servers is subsumed by the rule-based system.

We illustrate this last point with a couple of examples. Amoeba includes a server, the Swiss Army Knife (`SAK`) server, which makes specified RPC calls according to a time schedule. (The `SAK` server fulfills the same function as the Unix **cron** demon, but with greater flexibility.) This functionality could easily be handled by guarded commands which perform the desired actions. Using guarded commands permits actions to be initiated in response to much more than just the current time of day.

For example, we might have a system policy that states that a file system backup is to be started after 1 am as soon as the file system is "idle", for some notion of idleness; in any case the backup should be started no later than 5 am. This could be expressed by the following rule:

when ((1:00<*time*<5:00) **and** *idle*)
or (*time* = 5am))
and not *backed-up*
do *backup*

Rules may also be used to express the behavior of the Amoeba `boot` server, the server which ensures that other servers are available. Suppose we have some service, *Service*, for which it is the case that any member of the service may handle a request and running more instances of the service enables more requests to be handled. Using a rule-based system permits rules such as the following:

when SIZE(*Service*) < 3 **or**
Load(*Service*) > 1.0
do *Start another instance of Service*

This rule creates another instance of *Service* when the number of instances falls below three or when the load of the service as a whole goes above 1.0. Executing this rule at a replicated Meta service eliminates a problem with

the current **boot** server, namely, what happens when the **boot** server itself fails. (The current **boot** server is not fault-tolerant).

Amoeba also includes a **run** server which maintains a vector of processor loads, one entry for each processor on the network; this information is used to schedule tasks for execution. Instrumenting the distributed operating system and managing it with Meta enables Meta to directly carry out load-sensitive task scheduling. Moreover, a rule-based system enables complex policies such as processor affinity to be expressed.

The approach taken by Meta for system management makes many of the significant concepts of distributed systems theory readily accessible for use in system management. For example, the implementation of Meta exploits replication techniques and causally-ordered atomic broadcasts to control the system in a fault-tolerant and consistent manner. Moreover, the Meta approach makes unified management policies feasible for the control of modular, client-server based designs.

The ideas presented here will be tested out by using the Meta toolkit for system management tasks in Amoeba. The Meta system is currently being ported to Amoeba while work is concurrently underway at defining a high-level language for specifying control policies.

Acknowledgements This work has benefited greatly from discussions with members of the ISIS project at Cornell University. Meta was originally developed while the author was at Cornell. A high-level control language is jointly being developed with members of the ISIS project. Leendert van Doorn, Frans Kaashoek and Andrew Tanenbaum provided valuable feedback in the writing of this paper.

References

[1] K. Mani Chandy and Jayadev Misra. *Parallel Program Design*. Addison Wesley (Reading, Mass.). 1988.

[2] Marc Guillemont, Jim Lipkis, Doug Orr, and Marc Rozier. A second-generation micro-kernel based UNIX: Lessons in performance and compatibility. In *Proceedings of the USENIX Winter 1991 Conference*, pages 13–21, 1991.

[3] Keith Marzullo, Robert Cooper, Mark Wood, and Kenneth P. Birman. Tools for distributed application management. *IEEE Computer*, 24(8):42–51, August 1991.

[4] Keith Marzullo and Gil Neiger. Detection of global state predicates. In *Proceedings of the Fifth Workshop on Distributed Algorithms and Graphs*, October 1991.

[5] Keith Marzullo and Mark Wood. Tools for monitoring and controlling distributed applications. In *Spring 1991 Conference Proceedings*, pages 185–196. EurOpen, May 1991.

[6] Sape J. Mullender, Guido van Rossum, Andrew S. Tanenbaum, Robbert van Renesse, and Hans van Staveren. Amoeba – a distributed operating system for the 1990's. *IEEE Computer*, 23(5):44–53, May 1990.

[7] Richard F. Rashid. Threads of a new system. *Unix Review*, 4:37–49, August 1986.

[8] M. Spezialetti and J. P. Kearns. A general approach to recognizing event occurrences in distributed computations. In *The Eighth International Conference on Distributed Computing Systems*, pages 300–307. IEEE Computer Society, 1988.

[9] A. S. Tanenbaum, R. van Renesse, H. van Staveren, G. J. Sharp, S. J. Mullender, J. Jansen, and G. van Rossum. Experiences with the Amoeba distributed operating system. *Communications of the ACM*, 33:46–63, December 1990.

[10] Mark D. Wood. *Fault-Tolerant Management of Distributed Applications using the Reactive System Architecture*. PhD thesis, Cornell University, December 1991.