Delegation: Uniformity in Heterogeneous Distributed Administration

Jean-Charles Gregoire
INRS-Telecommunications

Delegation: Uniformity in Heterogeneous Distributed Administration

Jean-Charles Grégoire – INRS-Télécommunications

ABSTRACT

We describe a distributed administration tool based on the concept of delegation. Management operations are lightweight processes created on a manager's machine, but executed remotely on specific targets. Various libraries are provided on target machines to cover the widest range of administration tasks such as modification of configuration files, recurrent, scheduled executions of programs or process monitoring. The result is a single, unified framework for the majority of administrative operations, and a distributed platform to facilitate its use in a flexible way. The language used to realize the platform is small, unobtrusive and robust.

Introduction

System administration has slowly evolved from the historical machine-based perspective to distributed environments. In the Unix universe, administration is still a blend of machine-based operations combined with distributed applications, typically based on a client-server model. The historical heterogeneity of interfaces of machine-based systems has increased in the process.

Tools are built to alleviate the administration burden. They however very seldom cover the whole spectrum of operations, focusing rather on a specific range of problems. They are applications, or programming languages and they contribute to the plethora of information an administrator has to deal with – a form of *cognitive overload*.

We introduce here a distributed management platform based on the concept of *delegation*[2]. Management operations are lightweight threads, downloaded from the manager's host machine to selected targets. Specific language features guarantee that the operation requested is meaningful. The first release of this platform is operational and exploited across the machines of our group at INRS-Télécommunications.

In this paper, we describe the requirements a distributed management platform has to meet, describe the tools typically used and their main features. We introduce the notion of delegation and show how it supports very naturally distributed administration. We present the language used to implement delegation, its salient features and the additions we made to it. We conclude with the discussion of some unresolved issues.

Management Activities

We distinguish between two different categories of administration operations: *static*, or slowly changing, and *dynamic*, or quickly changing.

Static administration

Static administration englobes the customization and tailoring of a system to a specific site's requirements, that is, system configuration. It is mainly done at system creation time, and updated when a change is required. The most frequent changes occur when user configuration or file system structures are modified.

Examples of configuration operations includes kernel modification, file system creation, and maintenance of a variety of files under the */etc* directory.

Most changes to static configurations are immediate, although kernel modifications require a reboot to be effective. Some applications may also have to be signaled, or restarted.

Dynamic administration

This form of administration is more focused on the dynamic behavior of the system: processes and their characteristics, user sessions or I/O performance in general. The relevant information is typically held in the kernel, and made available through system calls. In this case, it has the several consequences:

1. The information is potentially costly to acquire
2. The information is imprecise (e.g., ps listing)

Some dynamic information is also reflected directly in the file system, such as pending print jobs, or process information in some versions of Unix.

Distributed Administration Tools and Methods

As networks become more pervasive, a number of tools have appeared to support distributed administration. Some are proprietary commercial platforms, but many are now bundled with most operating systems. Tools like the Network Information Services[4] (NIS, formerly YP) or Hesiod[3] remove the management of file-based static configuration away from the individual system. Based on a client-server model, they provide a single point of update of the information, and one or more points of query, for optimization. These tools have typically focused on the distribution of static-type information.

When better mechanisms to support the creation of distributed applications became available, other tools have also been built to resolve specific static or dynamic problems: improved distributed print management systems (e.g., ISPIN), distributed batch queues (e.g., qbatch, CONDOR) or software distribution update mechanisms (e.g., DEPOT).

New commercial offerings such as Tivoli Systems' Management Environment[9] go further by providing an extendible platform supporting remote operations across an heterogeneous system. At this stage though, such tools are still heavy to use, resource hungry, and often cumbersome to manipulate. They are typically aimed at very large networks.

For services not supported by distributed applications, the administrator is left with the usual practice of establishing a connection to the machine he wants to work on, and use his favorite tools, such as **perl**[6] or the ubiquitous **sh/awk/sed** mix to perform the required operation. A recurrent operation will be captured in a program, to be executed on specific request, or at regular intervals using the **cron** mechanism.

Remote connections as superuser however raise some security concerns in systems where passwords are transmitted un-encrypted, or through the use of mechanisms such as **equiv** files.

The bottom line for the administrator is a mix of machine-based and distributed tools with different programming interfaces, overlapping but incomplete functionalities, or simply new, but potentially easier ways of doing the Same Old Thing.

We offer to this state of affairs our perspective of a *distributed, uniform, programmable platform* for system administration.

Delegate

We have developed a *delegate* approach to administration, with a *unifying* perspective on the various activities we have described above.

A delegate has the following features:

1. Uniform language interface across all platforms,
2. One language for all activities,
3. Interface to most static and dynamic configuration information,
4. Built-in scheduling.

We also have the following overall goals in focus:

1. Localization of management information,
2. Local information processing,
3. Fine grained configuration,
4. Reliability through sound semantics.

Whereas the current trend in distributed services consists of moving static information away from the workstation, we take the opposite perspective. Since the facilities to support static configuration already exist on all machines, and the operating system still supports them, we make use of them. Rather, we remotely control the updates of this information. All queries are local, rather than remote as in the distributed model. Interestingly, this makes the system more robust, since it is isolated from server failure,[1] but also allows the fine tuning of the configuration of the systems, unlike the ''all the same under the same domain'' approach of some distributed administration tools.

Dynamic information is acquired only when requested by delegated activities, and shared between threads. It is processed locally and algorithmic operations can be performed as required. For example, processes consuming too many resources can be re-niced, or killed. This is the reverse of the SNMP view of transferring data to a manager, which makes the decision and orders an action from afar. With delegation, responsiveness and reliability are improved.

Implementation

The delegate model has been implemented by extending a functional language with some specific features. There are several reasons for this decision. We did not want to design yet another programming language, and thus chose to start with an existing base. **Caml-Light** was chosen over more Unix-flavored languages such as **Perl** or **TCL**, because of its inherent simplicity, the simplicity of its implementation, its two-tiered implementation model, based on a bytecode, and the existence of well-defined semantics. We do not imply that more mainstream languages could not satisfy our requirements but only that **Caml-Light** was the most manageable starting point.

Caml-Light is a *pragmatic* functional language, i.e., with a limited notion of side effects,

[1] although only to the extent that the working files are held locally

derived from the ML language, developed at INRIA. It is strongly typed and modular. We have extended the basic implementation to support remote execution. More specifically, we have added:

1. Multiple threads of execution,
2. Cooperative and preemptive thread scheduling,
3. A runtime command language for thread management,
4. Dynamic library loading,
5. In-memory binaries management,
6. Inter-threads communications.

The runtime environment was modified to operate from a socket connection, rather than interactively. The garbage collector was also extended to support multi-threading.

Threads are compiled into bytecode on the administrator's machine, then transferred to the specific target where they are linked and stored. Interface definition files hold the type and the syntax of the operations available on the required targets to reduce the probability of runtime errors. Our goal is to make threads completely safe.

The runtime environment supports thread management. Threads' runtime execution is controlled and they are executed sequentially.

Ideally, a thread implements a single activity, such as the update of a file or the enforcement of some *policy*[7].

Support Required

Most support for administrative operations is provided by platform-specific libraries, with a generic interface. Libraries currently in use include a kernel access library and a user administration library. An interface to a SNMP library is in the works.

No change to the kernel was required although access to more precise information would obviously help in problem tracing.

Security

Security is, understandably, a major administration concern. Problems are worse in a distributed environment than on a single machine, as confidential information is transmitted, sometimes in un-encrypted form, across the network. Other concerns, such as spoofing, also emerge.

The typical answer to this problem is to make sure that only encrypted information circulates, and only authorized users manipulate the information.

Our delegate platform exploits the basic Unix protection mechanisms. The runtime listens to a protected port, and accepts sessions only from a corresponding privileged port.

This however provides only the typical ''all-or-nothing'' style protection. Only the superuser can use the communication, unless the management program is setuid'd, which defeats the whole purpose.

Caml-Light provides us with another interesting protection mechanism. This language uses the module concept for strongly typed modular compilation of programs. We can use the mechanism to provide different interfaces to different people. Protection hinges thus around two mechanisms: protecting the access to the management platform and restricting the operations that can be performed, and restricting the management programs which can be created by the different users.

We feel that we have enough latitude to implement secure access schemes, without unduly restricting access to the tool.

Let us note that the different library interfaces scheme is a quite interesting way to implement administration delegation. Combined with the Unix file protection system, it is possible to statically define which functionality any user would have access too. Any user, with access to the sources can also define a proper subset of the functions for other users, simply be writing an adequate interface.

Administrator Environment

The administrator has a simple environment to compose threads, compile and download them and control their activation and execution on remote targets.

Threads can send information back to the administrator machine through a log channel. There is no specific demultiplexing protocol at this stage. Threads must identify themselves, their host together with the data transmitted to allow demultiplexing and potential dynamic visualization on the administrator's host.

At this stage, a simple combination of **grep**ing and **tail**ing is enough to extract and present the relevant information.

Example

The program in Appendix A is an example of recurrent file system monitor. Each 45 seconds, all entries in /etc/fstab are processed, and the local file systems are examined. Two functions, *search_char* and *search_line*, provide general line scanning support.

This code is provided as an illustration of concepts and of **Caml-Light** and therefore does not attempt to be optimal. Ideally, getfsent should be used for portability and a list of the file systems should be maintained. The support routines should be in a general library, in scanf compatible form, although **Caml-Light** provides a much more flexible mechanism which is not demonstrated here.

Conclusions

We have described the features of a distributed administration platform based on delegation. We are currently experimenting with a first release of the system. Our experience so far shows the flexibility and the power of the delegation paradigm, mainly for dynamic administration.

It does not however cover all aspects of administration at this stage, and we need, for example, a special mechanism to transmit user-requested operations, such as a password change, to the administrator's site. We plan on expanding our model to integrate such operations in the context of *delegation of authority*. We also need to improve the quality of information threads can send to the administrator's environment.

The conviviality and the features of the administrator's environment will also receive further attention, to facilitate the administration of a large number of machines.

Author Information

Jean-Charles Grégoire is assistant professor at INRS-Télécommunications where he is involved in research activities in distributed systems and network management. He can be reached via snail-mail at INRS-Télécommunications, 16, pl. du Commerce, Ile des Soeurs, Verdun, Qc, CANADA, H3E 1H6. He can be reached electronically at gregoire@inrs-telecom.uquebec.ca

Bibliography

[1] X. Leroy, M. Mauny, ''The **Caml-Light** System, Release 0.5, Documentation and User's Manual'', INRIA, Sept. 1992.

[2] J.-Ch. Grégoire, ''Management with Delegation'', IFIP '93: AIP Techniques for LAN and WAN Management, April 1993.

[3] S. P. Dyer, ''Hesiod'', Usenix Conference Proceedings, Winter 1988, pp. 183-190.

[4] SUN Microsystems Inc., ''Network Information Services''.

[5] T. Christiansen, ''Op: a flexible tool for restricted SU access'', Proceedings of LISA III, 1989.

[6] L. Wall, R.L. Schwartz, ''Programming Perl'', O'Reilly and Associates, 1991.

[7] E. D. Zwicky, S. Simmons and R.Dalton, ''Policy as a System Administration Tool'', Proceedings of LISA IV, pp. 115-123.

[8] Network Working Group RFC 1157 ''A Simple Network Management Protocol'', May 1990.

[9] Tivoli Systems, ''Tivoli Management Environment'', 1992.

Appendix A: Sample Caml-Light Code

```
(***************************************************)
(*   Identify Local File Systems with High Usage   *)
(***************************************************)
#open "NEW_unix";;        (* unix function calls *)
#open "NEW_timing";;      (* recurrence mechanism *)
exception FS_NOT_LOCAL;; (* file system not local *)
exception Found of int;;
exception Next_loop;;
(* Find position of a character within a line.     *)
(*       Return value = position in line           *)
let search_char line c =
 try
  for i=0 to ((string_length line)-1) do
   if (nth_char line i) == c then raise (Found i)
  done;
  -1
 with Found i -> i
;;
(* Extract string between the first pair of `:` *)
let sub_line line =
 let pos_1 = search_char line `:` in
  if pos_1 != -1 then
   let pos_2 =
    search_char (sub_string line (pos_1+1)
       ((string_length line)-pos_1-1)) `:` in
    if pos_2 != -1 then
     sub_string line (pos_1+1) pos_2
    else
```

```
      sub_string line (pos_1+1)
       ((string_length line)-pos_1-1)
     else
        ""
;;
let process_fs fs =
 try
  let data = statfs fs in
    (* calculate percentage utilization *)
    let t = float_of_int data.fd_btot in
     let f = float_of_int data.fd_bfreen in
      let u = t -. f -. (t *. 0.1) in
       (* Maximum utilization set to 80% *)
       if (u >. 80.0) then (
        print_string
  ("\nhostname: system getting full: "^fs^"\n");
  print_endline s;
       )
 with sys__Sys_error s ->
  prerr_string ("\nhostname:statfs: "^fs^"0);
  print_endline s
;;
(* Extract file system names from "/etc/fstab" *)
(*       and check utilization               *)
let check_local_file_system() =
 try
  let in_fstab = open_in "/etc/fstab" in
   while true do
    try
     let in_line = input_line in_fstab in
      (* Only local file system names *)
       if (search_char in_line `@`) == -1 then (
        (* Extract name and check utilization *)
        let fs = sub_line in_line in
         process_fs fs
       )
    with End_of_file -> close_in in_fstab;
     raise Next_loop
   done
 with sys__Sys_error s->print_endline s;exit 1; ()
    | Next_loop       ->flush std_err; ()
;;
(* Every 45 s., check utilization of file system *)
while true do
  print_string "\nhostname:Checking file system.\n";
  check_local_file_system();
  print_string "\nhostname:Done.\n"; flush std_out;
  delay 45
done
;;
```