

Monitoring and Instrumentation of a Clustered VoD System

Miguel Barreiro, Victor M. Gulias, Juan J. Sanchez
 {enano, gulias, juanjo}@lfcia.org
 Erlang User Conference, 3rd October 2000

1 Introduction

A Video-On-Demand (VoD) server is a system that provides video services in which a user can request a specific piece of video information (film-on-demand, remote learning, home shopping, interactive news...) at any time. The growing demand of such services suggests the design of scalable VoD servers, both in storage capacity and bandwidth. A hierarchical storage system (figure 1), based on a cheap Linux cluster (figure 2), is proposed to meet these requirements. At design time, the different architectural parameters (bandwidth and storage capacity at each level) and the operational procedures (admission policy, scheduling, file replacement and bandwidth assignment in the system) must be taken into account.

The concurrent and distributed language Erlang has been chosen for programming the prototype as well as for most of the modules of the final system. Erlang is a functional language developed by Ericsson, that has suitable features for the implementation of soft real-time fault-tolerant distributed processing systems. For example, some development tools, some libraries for the creation of graphical user interfaces, and even a distributed database, have been used for this project.

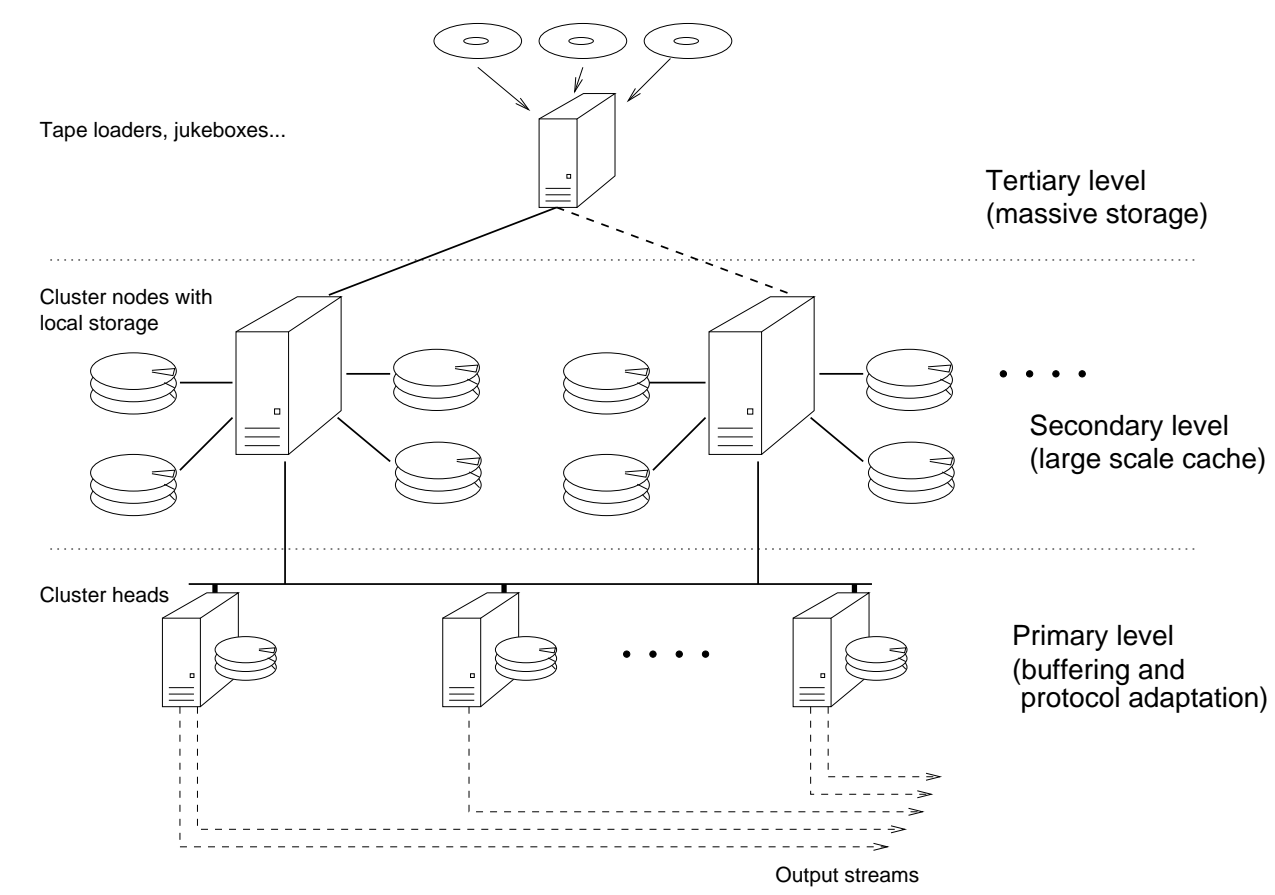


Figure 1: VOD Server Hardware Hierarchy

MONET is a simple and flexible monitoring tool, suitable for network monitoring, host performance tracking, and for the instrumentation of complex systems, among others. It was originally developed as a debugging and monitoring aid for a video-on-demand server under development at the LFCIA lab, and then evolved and generalized.

The whole foundation for MONET is the Erlang/OTP [1] platform, including MESH, EVA, SASL and Mnesia. Erlang has been chosen for its robustness features, ease of integration with the rest of the system, speed of development, and the functional background of many LFCIA members.

MONET was initially intended as a replacement for simple monitoring tools as MON [5], with additional support for variable tracking and graphing, as well as more complex instrumentation features.

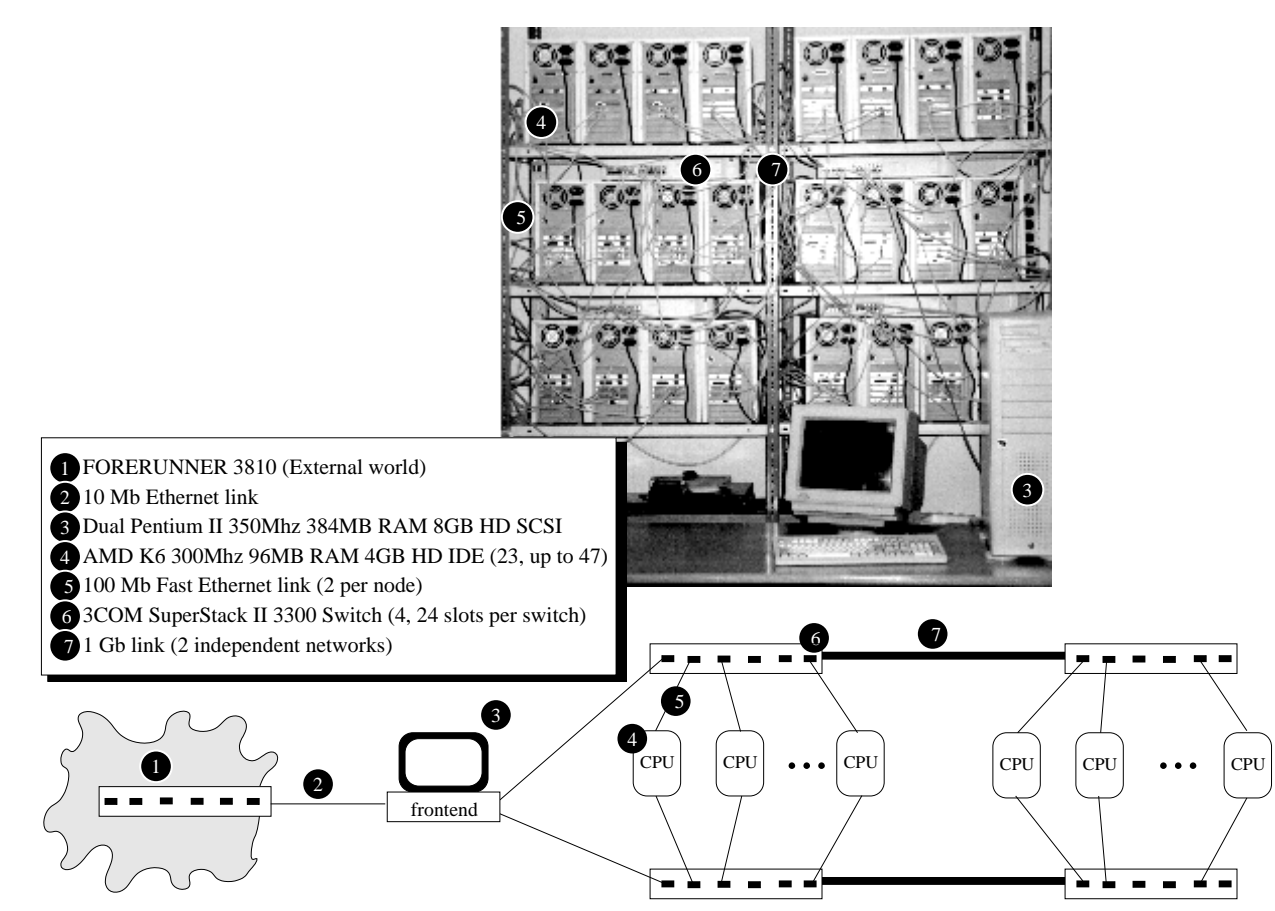


Figure 2: The Borg

2 The MONET Subsystem

2.1 Overview

MONET follows the MESH and EVA design, leveraging the infrastructure they provide and extending it. Figure 3 presents the modular structure of the system. For each measurement an MO (measurement object) is created, supervised by an MRP (measurement responsible process). A master handler (main_handler) receives MESH and EVA events and alarms, and may invoke specific action handlers according to complex configured conditions. Event and alarm logging is performed through the standard MESH logs; additionally, custom loggers can be plugged into main_handler or directly into EVA. The user interface is implemented through a web adaptation, using ERLATRON (see section 3) and INETS from erlets.

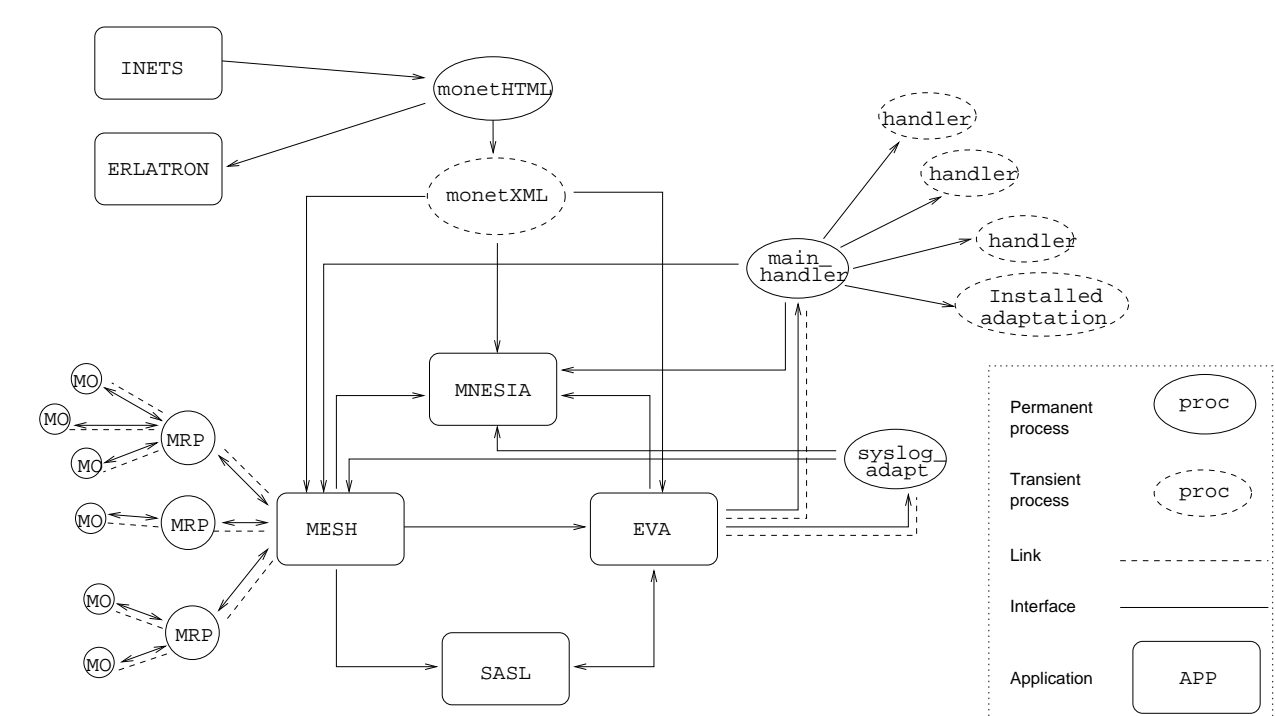


Figure 3: MONET structure

2.2 The Classes Tree

Instead of defining MESH Measurement Objects one by one, MONET creates a tree of monitored object classes, together with the measurement objects that will take objects of that class as monitored resources, like the one shown in figure 4. Each object class — a tree node — can be further specialized as desired.

Leaf nodes usually represent individual hosts or resources, but are not special in any way. The tree is traversed from the root towards a node to determine the measurement classes suitable for that node.

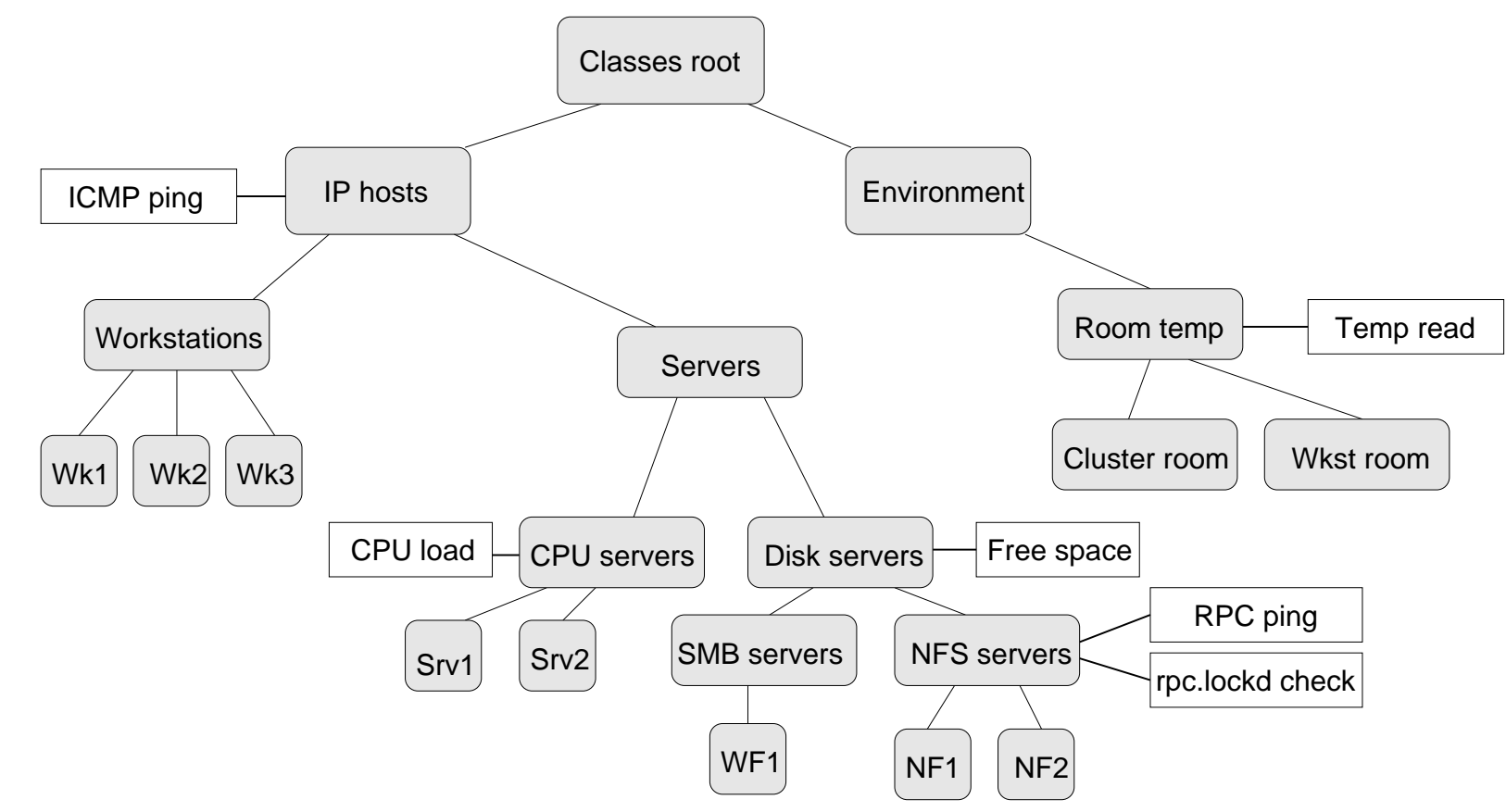


Figure 4: A sample classes tree

2.3 The Organization Tree

Resources to be monitored are defined as a directed graph, reflecting the logical grouping as managers see them. This logical grouping is completely artificial, and can be based on their physical, topological or simply organizational structure. Note that, despite its name, this structure is a directed graph and not strictly a tree, because branches can merge at any point (see figure 5 for an example). It is supposed to make sense to humans and has no other constraints, and does not even require all monitored resources to be present.

Each resource can have an attached list of classes; these classes position the resource in the classes tree, thus implicitly declaring the measurement objects that will monitor it.

Resources can also contain additional data in order to store their physical position, network connections, desired graphical representation, etc. This information can then be used by measurement objects to get additional configuration or by the user interfaces when representing the tree.

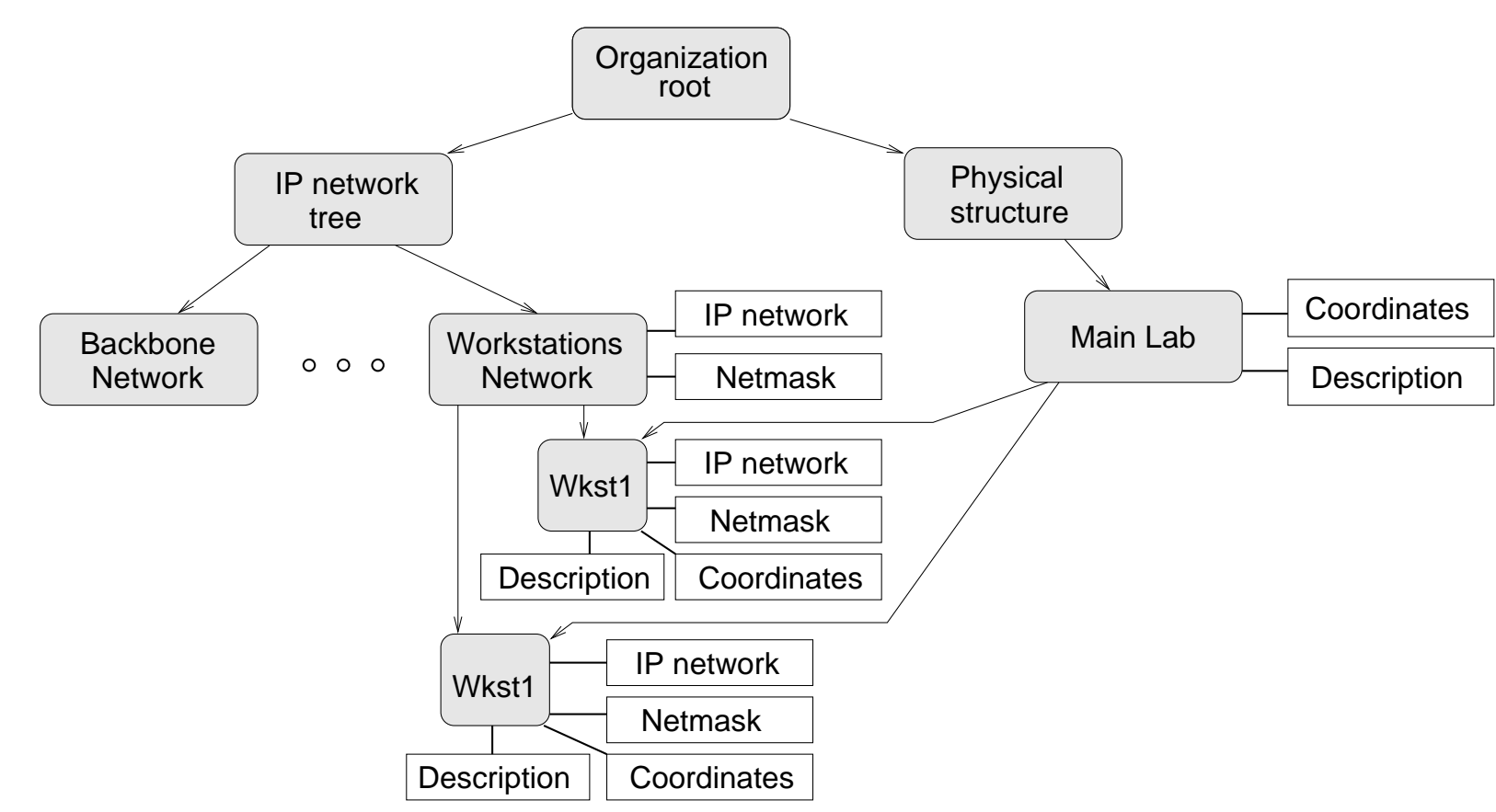


Figure 5: A sample organization tree

2.4 Measurement Objects

Following the MESH conventions, specific monitors are Measurement Objects, supervised by a Measurement Responsible Process. Measurement Objects can run in remote nodes.

In order to ease the development of simple, common monitors, MONET provides two generic measurement types which act as bridges between MESH and Erlang functions or Unix executables, respectively. Thus, existing monitor scripts and executables — such as those from MON — or Erlang code can be used as monitors without additional coding.

2.5 Alarm Handlers

Alarm handlers are called from main_handler when the appropriate alarms and conditions are triggered. Again, they can be either external executables or Erlang functions.

2.6 Alarm Destinations

Whenever an event or alarm is received, MONET checks whether it should call a handler. Alarm destinations are currently defined as functional expressions in order to define complex conditions, such as:

```
[
  % Call dumphandler whenever an alarm is received.
  % (i.e., when atdumps/always/5 returns true)
  { always, dumphandler },

  % Call myhandler if the sender is enano2@borg
  { {sender, [enano2@borg]}, myHandler },

  % Call thisHandler if extmodule:extfunc/5
  % returns true (called as
  % extmodule:extfunc(Name, Sender,
  % severity, class, {LocalArgs}))
  { {extmodule,extfunc,{localargs}},thisHandler },

  % Call dumper if either always()
  % or never() are true:
  { [any, [ {always,[ ]},{never,[ ]} ]], dumper },

  % Call selHandler if the two former are true:
  { [ all, [ [any, [ {always,[ ]},{never,[ ]} ] ],
    {extmodule,extfunc,{localargs}} ] ],
    selHandler }
]
```

This allows for composition of arbitrarily complex conditions as long as they can be expressed in Erlang. The use of higher-order functions to express complex configuration conditions is also employed in stylesheet selection (section 3.3).

2.7 XML Interface

The main MONET user interface is implemented as a set of erlets called from INETS. They never produce HTML output directly, but call the appropriate XML interface functions and transform the result instead, using ERLATRON (see section 3).

As long as MONET information can be retrieved from different devices, it is advisable to provide a general mechanism to adapt this content to the specific features of such output devices. Contrary to when style information is hard-coded into the content, separation of style from content allows for the same data to be presented in different ways. This enables:

- Reuse of fragments of data,
- Multiple output formats,
- Styles tailored to the reader's preference,
- Standardized styles,
- Freedom from style issues,
- Easy interface with third-party software.

In order to achieve such goals, MONET generates most of its results as XML documents which can be transformed at a later stage according to an XSL stylesheet.

3 The ERLATRON Subsystem

3.1 Overview

To perform the XSL transformation, MONET uses ERLATRON, a distributed XSLT processor implemented in Erlang using a C++ library (SABLORON [4]).

Figure 6 presents the actors involved in the ERLATRON subsystem. ERLATRON adapts the SABLORON library by using a port which performs basic transformations of a couple of Erlang binaries representing the XSL stylesheet and the XML source. SABLORON is based on the EXSLT library [3] and has been designed for performing fast and compact transformations. The port is managed by a generic server that offers XSLT services to any client. This slave server constitutes the basic processing unit, and considering the CPU cost of performing XSL transformations, low additional overhead is expected when used from Erlang.

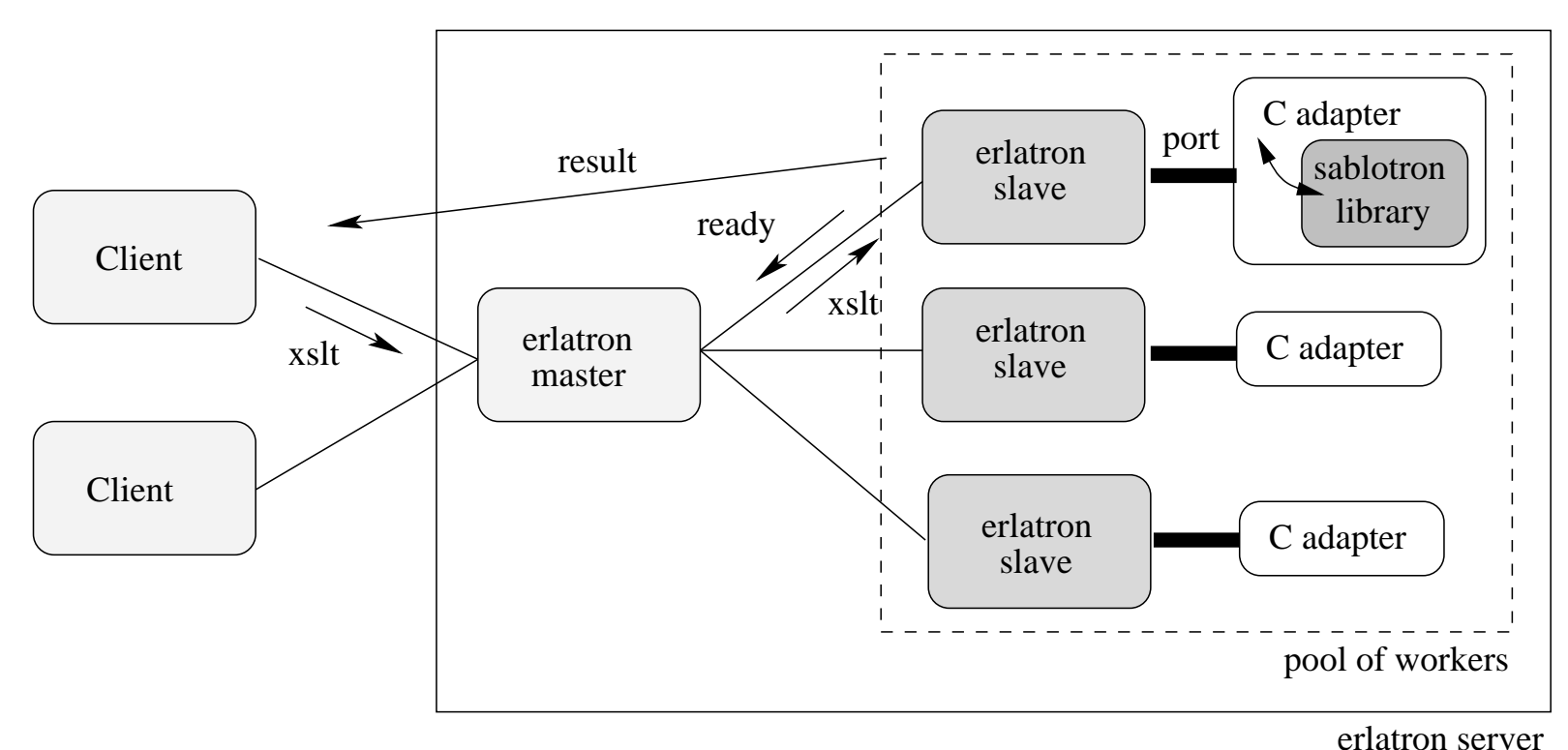


Figure 6: ERLATRON actors

In order to exploit a distributed framework, such as the Beowulf cluster introduced in [2], a simple master/slave architecture is deployed. In this setup, a master server is used to distribute requests to different slave servers running on a pool of computer nodes. The state of the master server is a collection of pending transformations as well as the information about idle slaves. The dispatching of requests, carried out by the master scheduler, consists of pairing a pending transformation with an idle slave server. Figure 7 shows the interaction among actors when solving an XSLT service.

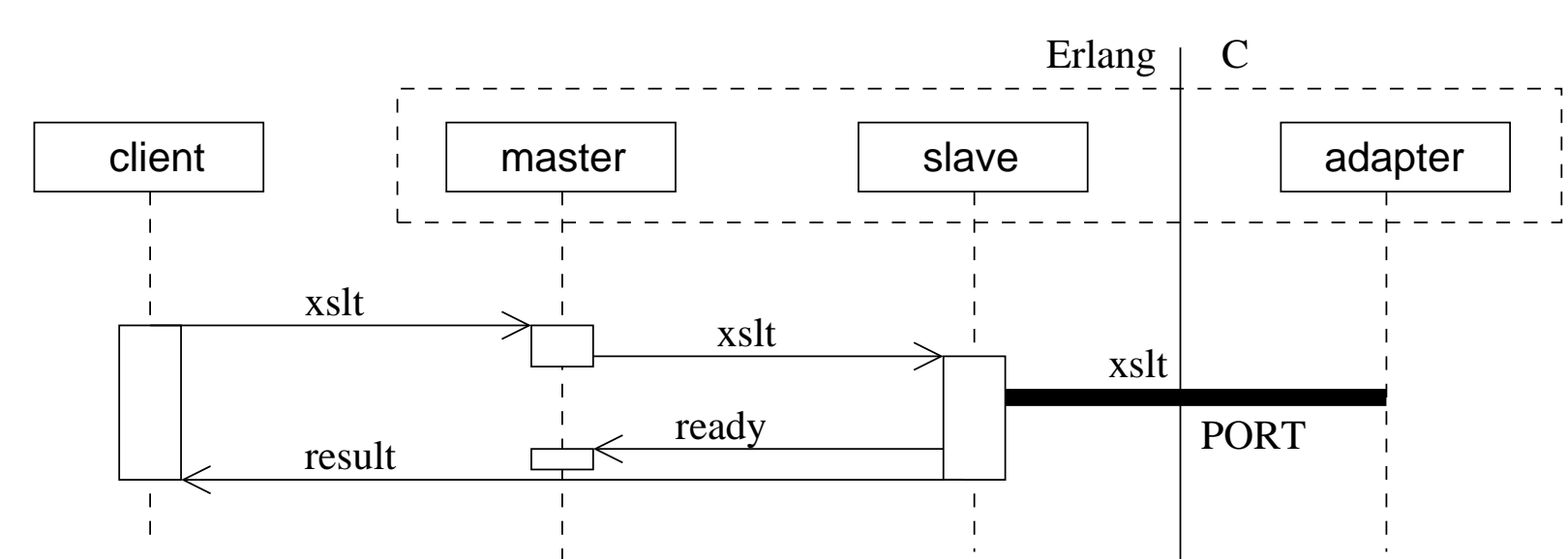


Figure 7: Interaction among actors

Clients interact with the master server through a global registered name, making an ERLATRON server takeover possible in case of failure without disturbing client interaction.

3.2 Benchmark

As the reader can guess, ERLATRON architecture seems to be quite interesting for web sites with a high number of requests which involve many XSL independent transformations for dynamic content generation. We are going to present some preliminary results when using ERLATRON on a Linux Beowulf cluster with a frontend and up to 22 nodes (Borg, figure 2). The frontend (Dual Pentium II 350MHz 384MB) runs INETS, serving a 64KB HTML generated using a 25KB XML document and a small XSL stylesheet; each node (AMD K6 300-266MHz, 96MB) hosts an ERLATRON slave. All the nodes are linked using a switched 100Mb Fast Ethernet.

Figure 8 shows the requests per second achieved when running Apache Bench (ab), a tool for benchmarking an HTTP server, at the frontend. As new XSLT processors are added to the slave pool, the server is able to increase its service rate until the concurrency level (C, number of simultaneous requests, -c) matches the pool size. Figure 9 presents the time taken to attend a collection of requests, varying the concurrency level.

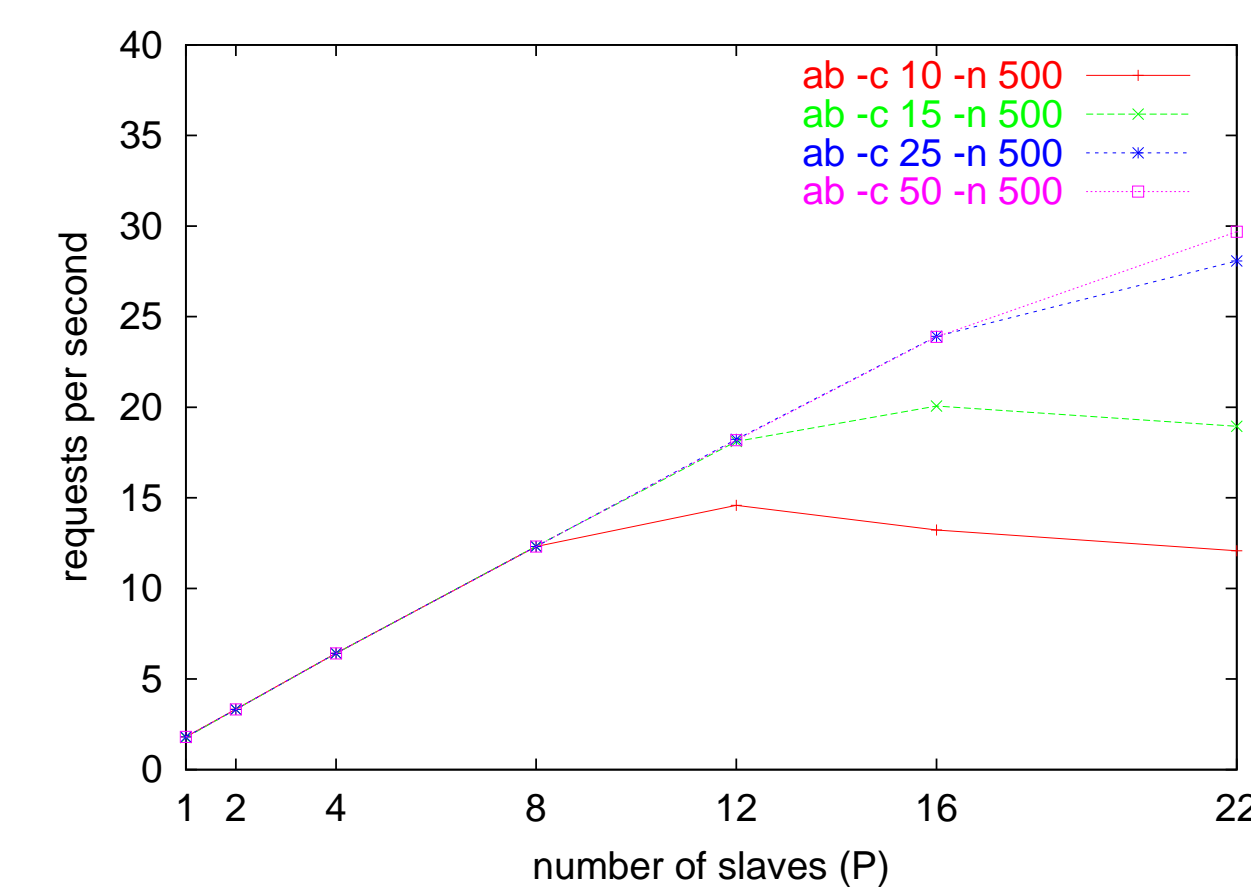


Figure 8: Requests per second

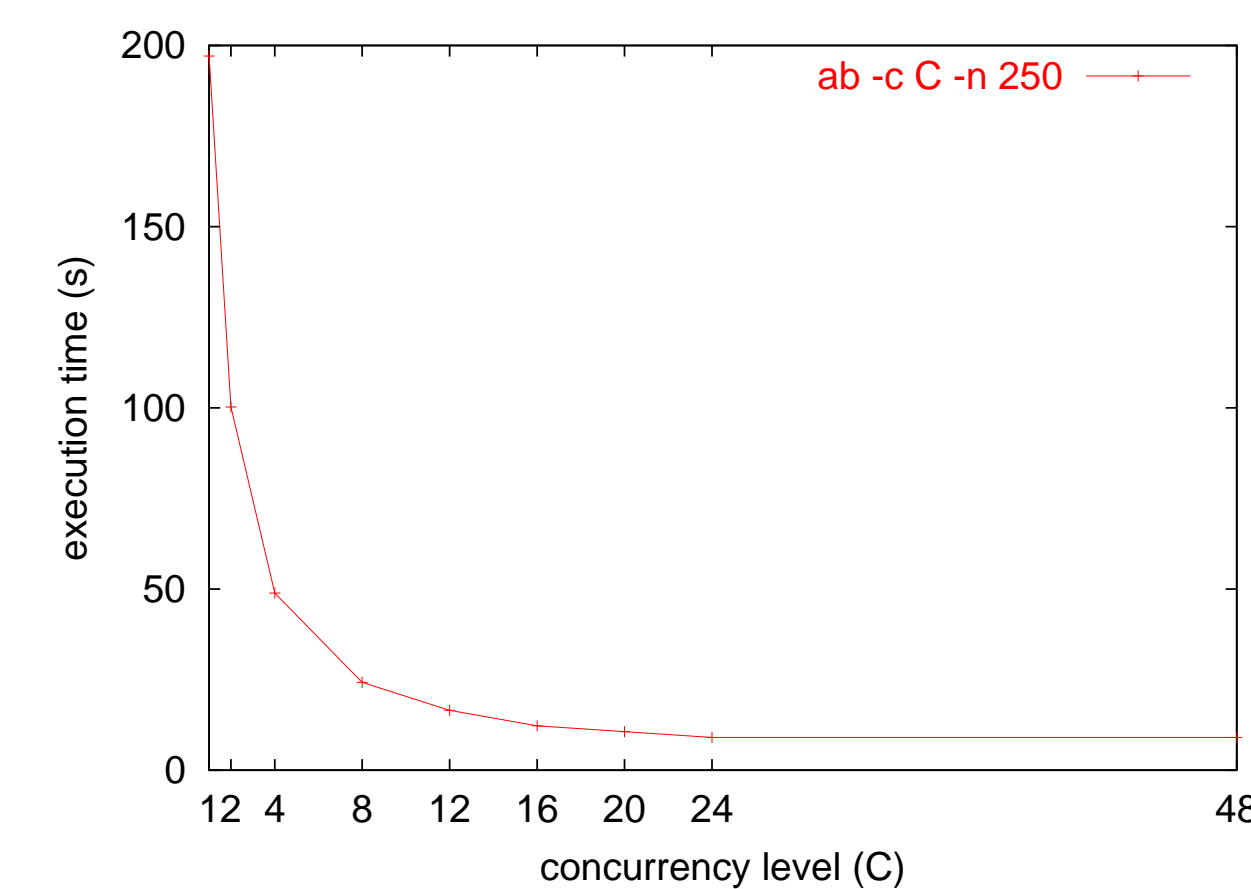


Figure 9: Execution time

Figure 10 shows how the average time for a request can be kept almost constant while concurrency level is not greater than the number of slaves. It should be pointed out that the overhead for the case $slaves = 22$ and C low is due to the use of nodes 17-22 which are AMD K6 266MHz, while nodes 1-16 are AMD K6 300MHz. The master scheduler should take this fact into account and introduce some kind of priority among slaves. This is particularly interesting when some slaves run at the frontend because that will reduce network communications.

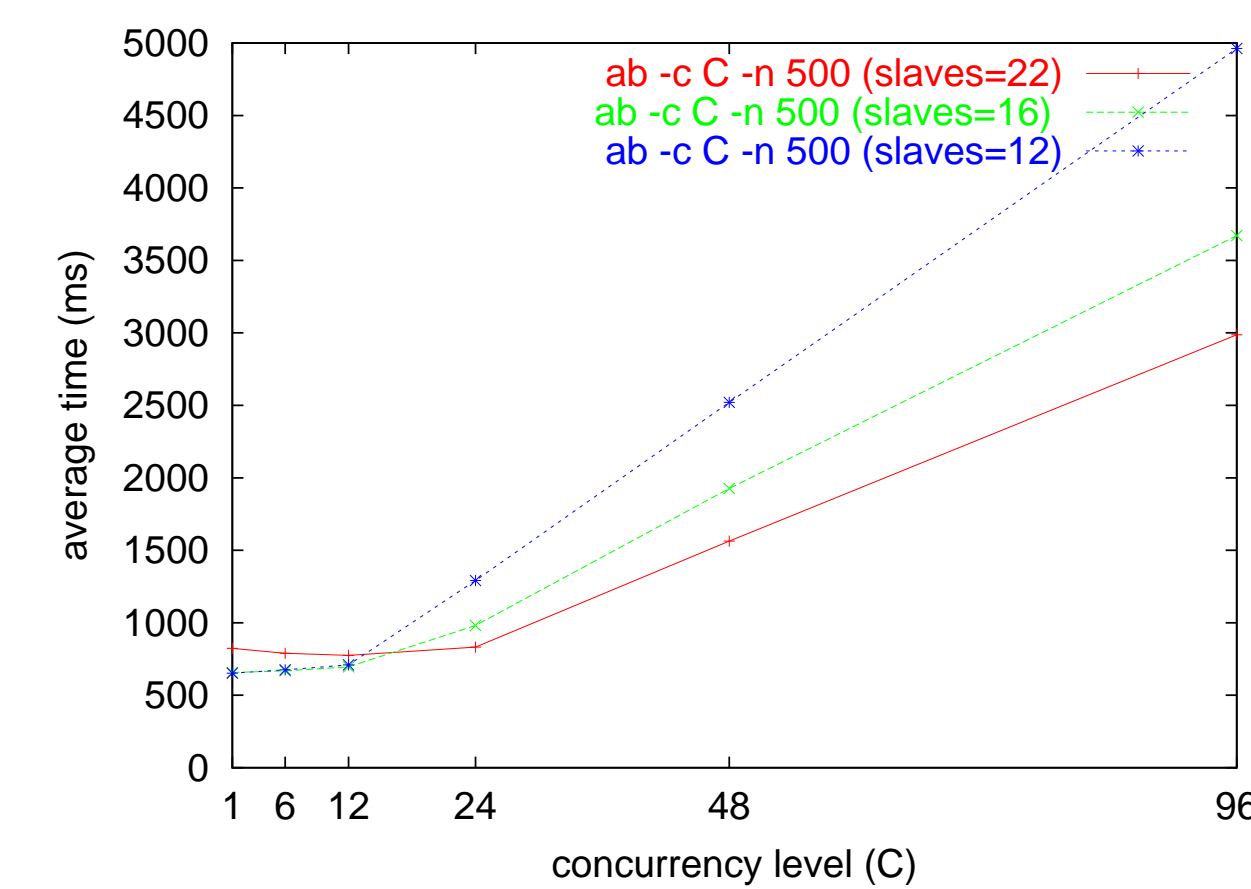


Figure 10: Average service time

3.3 Stylesheet Selection

Given a valid XML document, one of the ERLATRON duties must be the selection of an appropriate XSL stylesheet based on the type of the requested document and the nature of the output device, even though other parameters can be taken into account such as system workload (e.g. a simpler formatting can be chosen under heavy load condition). In order to accomplish this requirement, ERLATRON subsystem incorporates a module with two basic features:

- **Stylesheet registration:** A Mnesia database is used to store the binding between a document type and an XSL stylesheet. As the same document type can have different stylesheets, the database also stores a rule that decides whether the style can be applied.
- **Stylesheet selection:** Using the information stored in the XSL repository, a stylesheet, matching its rule in the given context, is chosen.

A rule for a given stylesheet is defined as a function that takes a context and an extra argument and returns true if the stylesheet can be applied to the document in such context. The context is defined as a list of pairs (Key, Value), typically the result of parsing the QUERY_STRING or the environment of an HTTP query. Some useful functions are provided to test for an specific (Key, Value) pair (equal) or for matching a regexp (match). For example, a rule to trigger an stylesheet when using the Lynx browser is lynx, defined as:

```
lynx(Context, _Extra) ->
  match(Context, {"User-Agent", "Lynx"})
```

Higher-order functions are used to define complex queries combining simpler rules. For example, the rule all takes a list of rules (functions and, optionally, the extra parameters) and tests all the conditions. As an example, the following Erlang term binds the document type meas with the XSL <http://my.host/xsl/style.xml> when using Lynx and a simple style is required.

```
{meas,
 "simple lynx style for meas",
 "http://my.host/xsl/style.xml",
 [all, [lynx,
 {equal,{'style', 'simple'}}]]}.
```

4 Examples

The following small example shows a simple alarm list browsed from Netscape, Lynx, and a WAP emulator, all with different styles.

The initial XML was, in all cases:

```
<ALARMS>
<ALARM>
  <NAME>ping_borg24</NAME>
  <SENDER>borg24</SENDER>
  <CAUSE>unknown</CAUSE>
  <SEVERITY>major</SEVERITY>
  <CLASS>equipment</CLASS>
</ALARM>
<ALARM>
  <NAME>cluster_temp</NAME>
  <SENDER>environ_mon</SENDER>
  <CAUSE>Possible air cond failure</CAUSE>
  <SEVERITY>major</SEVERITY>
  <CLASS>environmental</CLASS>
</ALARM>
<ALARM>
  <NAME>response_time</NAME>
  <SENDER>adapt_module</SENDER>
  <CAUSE>Service queue length too long</CAUSE>
  <SEVERITY>minor</SEVERITY>
  <CLASS>processing</CLASS>
</ALARM>
</ALARMS>
```

If the User-Agent is Lynx, the style chosen is a very simple one, without tables (figure 11).

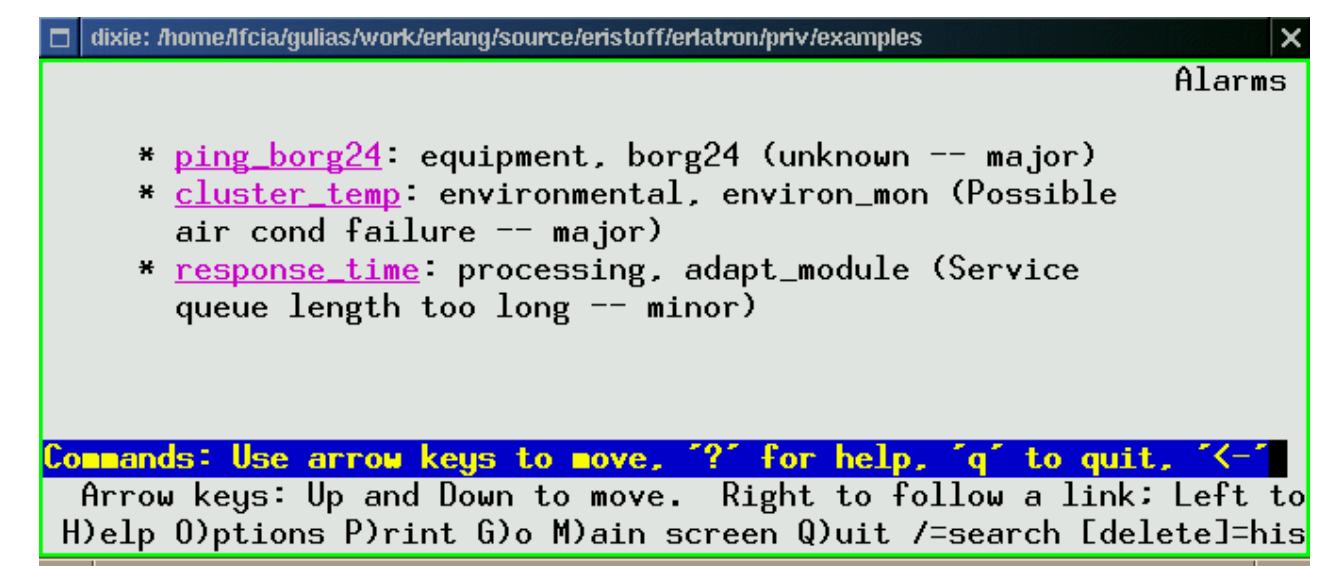


Figure 11: Alarms browsed from Lynx

In the User-agent is Mozilla (Netscape) instead, a richer XSL is chosen, displaying the alarms as a table (figure 12).

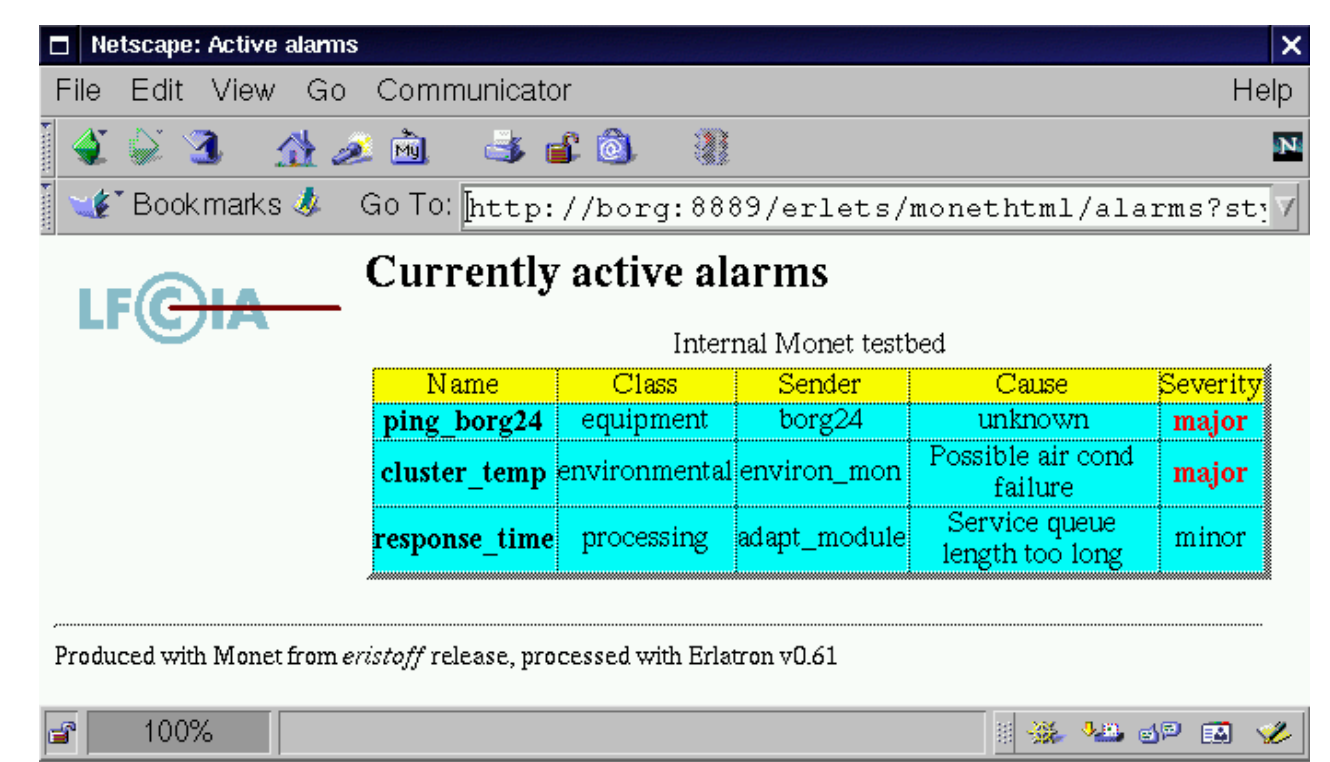


Figure 12: Alarms browsed from Netscape

Figure 13 (left) shows a compact alarm list produced for a WAP terminal. In order to speed download time, specific alarm details are included as WML cards in this WAP style, as shown in figure 13 (right).

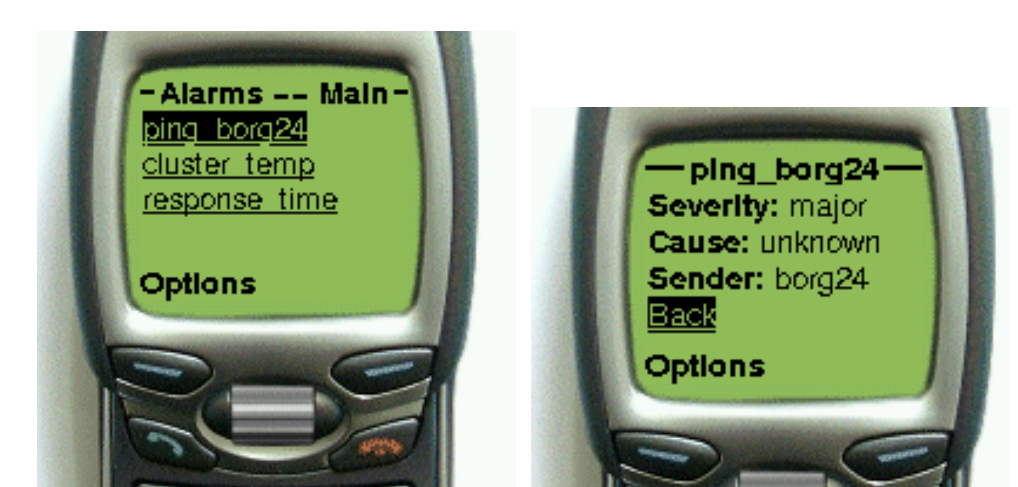


Figure 13: Alarms browsed from a WAP terminal

References

- [1] J. Armstrong, M. Williams, and R. Virving. *Concurrent Programming in Erlang*. Prentice-Hall, Englewood Cliffs, NJ, 1993.
- [2] M. Barreiro and V. Gulias. General Issues on Cluster Administration. In Rajkumar Buyya, editor, *High Performance Cluster Computing*, volume 1. Prentice Hall, 1999.
- [3] J. Clark. *Expat - XML Parser Toolkit 1.1*. <http://www.jclark.com/xml/expat.html>.
- [4] T. Kaiser. *Sabltron*. Ginger Alliance Ltd. <http://www.gingerall.com>.
- [5] J. Trocki. *mon, Service Monitoring Daemon*. <http://www.kernel.org/software/mon/>.