XML-based monitoring and its implementation in Perl

Marek Kamiński[1]

Abstract: This paper outlines in general, the problem of information systems monitoring and afterwards describes monitoring system developed in Lufthansa Systems Company for its customers information systems monitoring. It is focused mainly on XML-based mechanism developed for this system, but it explains also briefly others of its working principles. Also fundamentals of XML processing in Perl language have been shown and some very simple component designed for developed system has been presented. At the end of this paper, very short summary, small conclusions, proposals for interesting and useful extensions for the system and directions of future work and research have been outlined.

Keywords: XML, information systems, monitoring.

1. Introduction

Our present world can be characterized by very fast and progressive informatization of almost all domains of our lives. Number of information systems increases drastically and they provide more and more newer functions. However, despite that huge progress, almost every more complicated system still requires some kind of supervision, control and needs repair from time to time. Many systems have to be up and running all the time and their unplanned downtimes are unacceptable. This caused that many companies offer their customers - among other services - technical support, assistance and taking care of their IT systems for 24 hours per day and 365 days per year. This, in turn, requires at least efficient monitoring because we can never be sure that systems work properly without taking periodical measurements of their states. Additional problem is caused by the fact that very often, considered systems are geographically distributed and environments requiring monitoring are not homogeneous.

2. Problem characteristics

In case when approximately ten to twenty independent IT systems require monitoring and supervision, it is yet thinkable that such service can be provided manually. In case when number of such IT systems is much higher, automation and centralization of monitoring should be considered. Mentioned situation has been outlined in the figure 1:

[1] Gdańsk University of Technology, Faculty of Electronics, Telecommunication and Informatics, ul. Narutowicza 11/12, 80-952 Gdańsk Wrzeszcz
Lufthansa Systems Poland, Sp. z o.o., ul. Długie Ogrody 8, 80-765 Gdańsk
e-mail: {marek.kaminski}@eti.pg.gda.pl

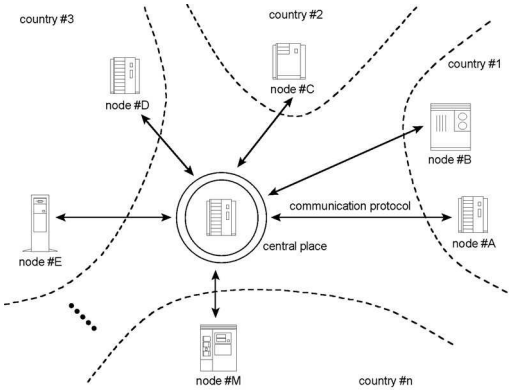

Figure 1. General view on the problem

This situation can be briefly described as follows:

- there are many independent nodes (machines) and their number is changeable in time
- many of them are working continuously
- they differ a lot (different hardware, different operating systems, different roles)
- they are geographically widely distributed
- all of them are accessible by network from one, central place

We can isolate three different elements:

- nodes
- communication protocol
- central place

Each of them has its specific character and brings other problems and other requirements. This is why proposing such centralized monitoring system is not trivial. Many factors should be taken into account. The following subsections present a deeper view on all of those three elements:

2.1. Nodes

Each node is usually different. Nevertheless, despite the differences we can always isolate some general things regarding all of them. One of such things is a kind of resource. Each node has only two of them. It has physical resources (e.g. processors, memory, disks) and logical resources (e.g. processes, data). Logical resources depend on physical ones and they can be considered as software and hardware as well. It has been shown in the figure 2:

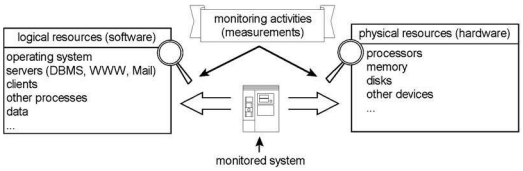

Figure 2. Monitored system

Undoubtedly monitoring activities should concern both kinds of resources. Because many resources have their own, unique characteristics, monitoring activities regarding them also differ. Some measurements should be performed nearly continuously, while some of them should be performed periodically or occasionally. Generally saying, frequency of measurement should be adjusted to the frequency of phenomenon that is measured. All those reasons cause that automation of monitoring should allow for different monitoring activities, because otherwise it rather might not be useful. Other important thing is overhead related to monitoring. Situations when acts of measurement change monitored environments parameters significantly are unacceptable.

2.2. Communication protocol

Communication protocol is other important and non-trivial element of centralized monitoring system. It has been schematically shown in the figure 3:

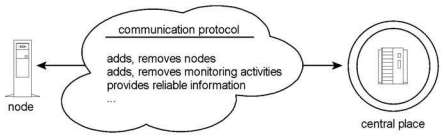

Figure 3. Communication protocol

That protocol should have at least the following characteristics:

- be able to add or remove any node
- be able to add or remove any monitoring activity at any node
- be able to transport to central place different results of different measurements
- ensure reliability of information (avoid distortions, e.g. when node is unreachable)
- be fast enough to fulfill system goals and do not cause noticeable network load
- be failure resistant - be able to handle any extraordinary situations

We can of course enumerate other characteristics which such protocol should have (e.g. security, working in two directions) but those listed above are absolute minimum, because without them implementation of centralized monitoring system which is able to measure many characteristics at many nodes can be really difficult if not impossible.

2.3. Central place

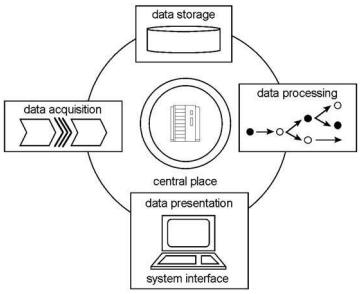

Figure 4. Central place

Figure 4 shows the monitoring system core that has a lot of work to do all the time. If system has to be useful we may suspect that it should work almost in a never-ending loop and perform at least the following tasks:

- acquire data from remote systems which need monitoring
- process that data and mine essential information from it
- store that data (for potential further processing and evidence purposes)
- present it in understandable form (communicate it in all necessary ways)

It is important to mention here that system should be able to manage with rather high volume of data, so should be able to process it efficiently and be fast enough. Other essential thing which should be taken into account during development of such system (and which can be concluded from previous subsections) is variety of measurements and thus also variety of forms which their results have. The system should be able to interpret any results sets. Such capability is needed because unprocessed (and not interpreted) information is not worth much. System should be able to mine from data this what is essential in considered moment in time and to inform its operator about it. Ways of data presentation and communication should utilize available presentation and communication mediums efficiently and do not waste them.

3. Implementation

Problem outlined briefly in the whole second section occurred in the real world in Lufthansa Systems Company (which is the member of Lufthansa group) and has been solved in a very flexible way. Developed system uses the XML language to provide platform allowing many different nodes for communication with the central system server. XML has been also used for providing system with the capability to link dynamically results of measurements with their interpretations.

3.1. Short description of developed system

The system works in a client-server architecture and consists of one central server retrieving measurement results from many system clients attached to it. In order to ensure minimal load on monitored machines, monitoring activities have been reduced to only essential minimum. All analysis of gathered data is performed on the system server. Typical way in which MonJaMi system works can be summarized as follows:

- Each monitored node has MonJaMi system client installed on it. This client has its own and internal scheduler (what makes it independent of Unix **cron** program) that runs periodically (in moments defined in **crontab.txt** file) monitoring scripts which gather necessary information regarding monitored parameters. After gathering of necessary information is done, each monitoring script (called also component) formats results to Xml format. Each client can be assigned many components.
- Reports constructed in such way are afterwards sent to system server.
- System parser processes incoming reports periodically (e.g. every one minute) and basing upon them and upon (so called) configuration records it interprets them and performs proper actions. It informs for example a system operator (through the web interface) that values of some parameters exceeded their first critical thresholds or that values of other parameters returned to normal.
- All gathered and processed information are inserted into the system database.

As communication medium used for sending data from clients to server, the company has chosen SMTP protocol (so simply saying, clients send to server e-mails). Newer version of MonJaMi system (currently under development) is also able to use an alternative method: SNMP protocol, instead of SMTP. The current system interface is WWW interface (accessible in company intranet through browser applications), because such solution seems to be the most flexible and reasonable at this moment, nevertheless nothing stands in the way to replace it with some other interface, written for instance in JAVA or in other language.

3.2. Data formats

Xml in MonJaMi has two roles. One of them is to act as a link between many different nodes and the central system server. The second role is to provide mechanism allowing for linking measurements results with their interpretation dynamically. The first role was done by standardization of reports format. Since now, each report looks like this below:

```
<report>
  <hostname>testhost1</hostname>
  <type>SYS_LOAD</type>
  <time>1047545227</time>
  <load1>1.15</load1>
  <load2>0.73</load2>
  <load3>0.55</load3>
</report>
```

Each report has its root tag that is named `<report>` and each root tag should obligatory contain the following three tags, nested in it:

- `<hostname>` - containing the name of machine where it comes from
- `<type>` - containing the type of report (informing what measurements it contains)
- `<time>` - containing the time when the measurement was made (counted in Unix notation, so as the number of seconds elapsed since the beginning of year 1970)

They build a complete report header (sent in this example by component calculating system load), identifying the node and the measurement activity. Measurement results should be stored between any designer-defined and designer-named tags. Their number is not limited and they can be freely nested. This is possible, because mechanism responsible for reports reading simply skips their structure and passes the whole information (embraced in them) on to other system mechanism that performs further processing. That mechanism requires from component designer providing code that can process reports designated for it. In above example, designer-defined tags are flat (not nested) and are named: `<load1>`, `<load2>` and `<load3>`.

The second role of Xml in the MonJaMi system is related to its configuration. The system configuration (stored in Xml file) informs system, in fragments of Perl code, how it should process and interpret incoming data. Structure of that configuration file has been shown below:

```
<events>
  <include>
    ...
  </include>

  <event_process>
    ...
  </event_process>
  ...
  <event_process>
    ...
  </event_process>
</events>
```

Tag `<include>` is designated for storing in it those fragments of Perl code which are used many times. In practice, it stores many auxiliary functions, used in other fragments of Perl code, in fragments that are responsible for measurements results processing and interpretation and which are defined in tags `<event_process>`.

Each `<event_process>` tag (called configuration record) contains in (nested in it) `<err_condition>` tag, Perl code that should be used by system to process data from precisely defined reports. In order to identify reports to which defined code should be applied, that tag contains (also nested in it) tags `<hostname>` and `<type>`.

MonJaMi system extracts from each report values defined in its `<hostname>` and `<type>` tags, then search in configuration file for `<event_process>` tag with the same values of those two tags (nested in `<event_process>`) and if such configuration record exists, system executes Perl code included in its `<err_condition>` tag. That Perl code should take care of report processing appropriate to report type and to its contents, while MonJaMi system takes care of making the whole content of report accessible from the level of that Perl code. After report is processed,

that code should place results of processing into a complex data structure defined beforehand. MonJaMi will then be able to insert them into the system database.

Introduction of the mechanism and data structures described in this subsection made the MonJaMi system very flexible, extensible and independent of the data that it processes and the parameters that it measures. It means that even though that system comes with many predefined and useful database-related and system-related components, it supports the addition of completely new components. Example of some simple component has been presented in the section 4.

3.3. Diagram of XML processing

Processing of Xml data by MonJaMi system parser has been shown in below diagram:

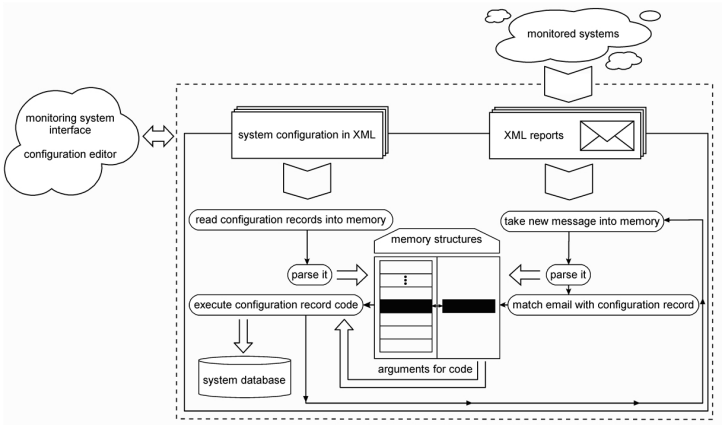

Figure 5. XML processing diagram

All Xml processing is done also in Perl language, with the usage of free `XML::Simple` library. Above diagram shows in more readable form how mechanism of dynamic linking works because it gathers all previous descriptions and information into one place. It should be read as follows:

1. MonJaMi configuration (written in Xml language) is read into memory
2. the Xml library parses it and stores it in the memory in much more useful form
3. new Xml report arrives
4. system reads it into memory
5. the Xml library parses it and stores it in the memory in much more useful form
6. system, basing on the values from `<hostname>` and `<type>` tags of the Xml report, matches report with Perl code which is able to process it
7. that code is executed by system and takes currently considered report contents (so measurement results) as arguments
8. results of processing are stored in the system database from where they are accessible for the system operator interface

9. procedure goes to the point 3

Above diagram can be extended but because I tried to focus in this article only on general idea of the system and on the XML processing in Perl, I omitted some less important issues but fundamental working principles have been presented accurately.

The following section describes more deeply the way of XML processing in Perl.

4. XML processing in Perl

XML processing in Perl is simple. Perl is high-level language and its data types allow programmers for building complex data structures easily. Conversions from XML data representation to Perl data representation or vice versa look very naturally.

4.1. Necessary minimum of Perl fundamentals

Perl has three built-in data types:

- scalars (strings, numbers, references to any data type)
- arrays of scalars (tables of scalars, indexed by numbers)
- associative arrays (hashes) of scalars (tables of scalars, indexed by strings)

Basing on both kinds of arrays and on references to them, programmer can build almost every required data structure. For instance, the tree data structure similar to structure of MonJaMi configuration file can be built as follows:

- scalar variable named `<events>` should be a reference pointing to the hash of scalars
- that entry of above hash which is identified by key `<include>` should be a string (which is a kind of scalar that can store bodies of auxiliary functions)
- that entry of above hash which is identified by key `<event_process>` should be a reference pointing to the array of scalars
- each scalar in that array of scalars should be a reference pointing to the hash of scalars
- each of those hashes should contain at least keys `<hostname>`, `<type>` and `<err_condition>` and those entries should be strings

After assuming convention that is used in Perl for accessing data in such complex data structures, we can show above example schematically as below:

```
$events = {
            include       => 'my $function1 = sub {...} ...',
            event_process =>
            [
              {
                hostname      => 'testhost1',
                type          => 'DBS_LOCKS',
                err_condition => 'here Perl code is stored...', ...
              },
              {
                hostname      => 'testhost1',
```

```
            type          => 'SYS_LOAD',
            err_condition => 'here Perl code is stored...', ...
          },
          ...
          {
            hostname      => 'testhostN',
            type          => 'NET_THROUGHPUT',
            err_condition => 'here Perl code is stored...', ...
          }
        ]
      };
```

Values in italics can be easily accessed in Perl with the use of the postfix arrow notation:

```
print $events->{event_process}->[1]->{hostname} . "\n";
print $events->{event_process}->[1]->{type}     . "\n";
```

It prints on the screen values `testhost1` and `SYS_LOAD` and such configuration record will be used by the system to process report given as example also in subsection 3.2. This chain of `->` operators can be of course longer and differs a lot if processed data structure is more complex. The only important thing which should be noticed here is that indexing in plain arrays should be done with the use of `[ ]` brackets, while indexing in associative arrays should be done with the use of `{ }` brackets.

To use outlined idea in practice, Perl developers prepared many more or less sophisticated libraries converting data from their XML representation to Perl representation. One of them, described here, is named `XML::Simple`. To use it in own scripts, programmer should download it and include it to scripts that will use it, using the following command:

```
use XML::Simple;
```

Since now, loading XML data, parsing it and putting it into the system memory, can be done really easy:

```
my    $xml = XML::Simple->new();
my $events = $xml->XMLin($configuration);
```

After execution of that code, XML data provided in `$configuration` variable (which can store either the name of XML file or the XML string) will be stored in complex data structure and reference to it will be stored in `$events` variable. Reversed operation can be done equally easily:

```
my           $xml = XML::Simple->new();
my $configuration = $xml->XMLout($events);
```

Because XML document structure can be much more complex than structures presented so far (for instance, XML tags can contain attributes, exactly as well known HTML tags), default behaviour of `XML::Simple` library can be easily changed, by

passing to it some control arguments. Those arguments and many other features of this library are described in deep details in its comprehensive documentation. One of interesting attributes is named noattr. If its value is set to 1, tags in XML structures generated by this library will contain no attributes. Any keys and values of associative arrays will be represented then as nested XML elements instead of XML elements with attributes. This particular example means that if we wish, we can read the following XML:

```
<data>
  <person firstname="John" lastname="Smith">
    <email>john.smith@yahoo.com</email>
    <email>j_smith@gmail.com</email>
  </person>
</data>
```

and save it as:

```
<data>
  <person>
    <firstname>John</firstname>
    <lastname>Smith</lastname>
    <email>john.smith@yahoo.com</email>
    <email>j_smith@gmail.com</email>
  </person>
</data>
```

4.2. Construction of sample component

The simplest version of `SYS_LOAD` component can look like this shown below:

```
use XML::Simple;
use Monjami::GlobalVar;    ## MONJAMI constants
use Monjami::Common;       ## MONJAMI functions (e.g. send_email)

my $report =
{
    type     => 'SYS_LOAD',
    time     => time(),
    hostname => `hostname` ## result of system hostname command
};                         ## (assume that it will be testhost1)
$output = `uptime`;        ## result of system uptime command
$output =~ /average:\s*([0-9.]+),\s+([0-9.]+),\s+([0-9.]+)/;

$report->{load1} = $1;
$report->{load2} = $2;
$report->{load3} = $3;
my $xml = XML::Simple->new();
send_email($xml->XMLout($report, noattr => 1, rootname=> 'report'));
```

The first three lines informs Perl that it should use `XML::Simple` library and special MonJaMi packages (with their functions and constants, such as IP address of mailserver). The following six lines of code prepare the `$report` data structure which will be converted into XML format and fills it with the type `SYS_LOAD`, with

the current time in Unix notation and with the name of the host that generates this report. The following line of code executes one of operating system command (uptime) that measures system load and stores its results in `$output` variable. The next line of code, using regular expressions mechanism, extracts from this variable three floating point numbers and stores them in variables `$1`, `$2` and `$3`. Those variables are then added to associative array pointed by `$report` reference as the values identified by keys `load1`, `load2` and `load3`. The last two lines of code convert `$report` data structure into Xml format and sends it to MonJaMi server using one of functions delivered with MonJaMi.

Reports generated by above Perl code could be interpreted for instance by below interpretation algorithm embedded in MonJaMi Xml configuration file:

```
<event_process>
<hostname>testhost1</hostname>
<type>SYS_LOAD</type>
<active>1</active>     <!-- component is active (it works)     -->
<timeout>600</timeout> <!-- it should receive data every 600 s -->
<err_condition>
<![CDATA[
## load thresholds:    load1  load2  load3         ## severity:
my @severity_map = ( [  12.0,  11.0,  11.0  ],     ## fatal    (3)
                     [  10.0,   9.0,   9.0  ],     ## error    (2)
                     [   8.0,   7.0,   7.0  ]  );  ## warning  (1)

my @load = ( $report->{load1}, $report->{load2}, $report->{load3} );
for my $s (0..2)
{
  for my $l (0..2)
  {
    $severity = 3 - $s if( $load[$l] >= $severity_map[$s]->[$l] );
  }
}
if( defined($severity) )
{
 $err_text = "Load too high: $load[0], $load[1], $load[2] !!!";
 $statistics->{"load1"} = $report->{"load1"}; #means: store it in DB
 $statistics->{"load2"} = $report->{"load2"}; #means: store it in DB
 $statistics->{"load3"} = $report->{"load3"}; #means: store it in DB
}
]]>
</err_condition>
</event_process>
```

This code tells MonJaMi that it should inform its operator that problem with load occurs using communicate **Load too high: *...load values...* !!!** . It also contains `SYS_LOAD` configuration matrix telling system what load values are unacceptable and allowing it to infer from them what priority (`$severity` variable) should be assigned to considered load problems. Lines `$statistics->{...} = ...` tells system which values should be stored in the database for statistical purposes, to allow system for generating from them charts, showing how parameters were changing in time.

5. Conclusions and future work

This article shows that Xml can be used efficiently as a platform for communication between different machines, with different environments, in order to solve concrete and important problem. It also shows the advantages that stem from good generalization. Lufthansa Systems ROC Helpdesk successfully uses described system for over four years. It saves a lot of time so administrators who are using it can focus on problems requiring more analysis and effort. Because some things can be still improved and some new interesting features can be added, MonJaMi system is still under development but its main idea and solid core stays unchanged.

Development potential of all monitoring systems in general so also of MonJaMi in particular is quite a big. They can for instance try to make prognosis and predictions regarding potential future problems (with the use of more or less sophisticated methods of artificial intelligence or inference), knowledge collected by them can be used to estimate hardware requirements for future customers, they can be fully integrated with internal company reporting and documents circulation systems and so on.

Other interesting idea is to propose more general monitoring system, which could be called "reactive". By this term I mean system, which will be extended with the ability to repair many kinds of detected errors and failures automatically, and my future work and research will head exactly towards this direction.

References

Jianwei, L., Hongbin, C., Pandeng, J. and Meirong, C. (2005) *Design and Implementation of Grid Monitoring System Based on GMA*. College of Computer Science, University of Electronic Science and Technology, China.

Kamiński, M. (2007) *Data replication and its monitoring.* Gdańsk University of Technology, Poland.

Kamiński, M. (2006) *Monitorowanie stanu systemów informatycznych na przykładzie aplikacji opracowanej w firmie Lufthansa Systems.* Gdańsk University of Technology, Poland.

Kamiński, M. (2005) *System monitorowania prac organów przedstawicielskich - doświadczenie z zakresu e-demokracji.* Gdańsk University of Technology, Poland.

McLean, G. (2006) *XML::Simple - Easy API to maintain XML.* `http://search.cpan.org/`

O'Neill, D. (2004) *Building Linux Monitoring Portals with Open Source.* Linux Today (number 12).

Škiljan, Z. and Radič, B. (2004) *Monitoring systems: Concepts and tools.* University Computing Centre, Croatia.

Ziad Albari, M. (2005) *A Taxonomy of Runtime Software Monitoring Systems.* Christian-Albrechts-Universität, Institut für Informatik, Germany.