LOCAL MONITORING AND CONTROL SYSTEM FOR THE SKA TELESCOPE MANAGER: A KNOWLEDGE-BASED SYSTEM APPROACH FOR ISSUES IDENTIFICATION WITHIN A LOGGING SERVICE*

M. Di Carlo, INAF Osservatorio Astronomico di Teramo, Italy
M. Dolci, INAF Osservatorio Astronomico di Teramo, Italy
G. M. Le Roux, SKA South Africa, Cape Town
R. Smareglia, INAF Osservatorio Astronomico di Trieste, Italy
P. S. Swart, SKA South Africa, Cape Town

Abstract

The SKA Telescope Manager (SKA.TM) is a distributed software application aimed to control the operation of thousands of radio telescopes, antennas and auxiliary systems (e.g. infrastructures, signal processors, ...) which will compose the Square Kilometre Array, the world's largest radio astronomy facility currently under development. SKA.TM, as an "element" of the SKA, is composed in turn by a set of sub-elements whose tight coordination is ensured by a specific sub-element called "Local Monitoring and Control" (TM.LMC).

TM.LMC is mainly focussed on the life cycle management of TM, the acquisition of every network-related information useful to understand how TM is performing and the logging library for both online and offline sub-elements. Given the high complexity of the system, identifying the origin of an issue, as soon as a problem occurs, appears to be a hard task. To allow a prompt diagnostics analysis by engineers, operators and software developers, a Knowledge-Based System (KBS) approach is proposed and described for the logging service.

INTRODUCTION

A log message is the simplest possible storage abstraction which says what happened and when. It is an append-only, totally-ordered sequence of records timestamped. So, a log is not all that different from a file or a table. A file is an array of bytes, a table is an array of records, and a log is really just a kind of table or file where the records are sorted by time.

Since it is a very simple concept, developers tend to underestimate the logging system but logs record *what happened and when* and, for distributed data systems, this can be the only way to find out the origin of an error.

Usually log files are written in a natural language (human readable) and, even if it is very common, this is not the best way to store informations: it does not allow to reason programmatically[**] about those information.

Building a logging system with a declarative language (for instance prolog) can give the possibility to reason about facts of the communications and operations with an inference engine. For the specific case this document is aimed to, with the adoption of a knowledge-based system approach, TM.LMC could give to SKA engineers, operators and software developers the possibility to ask high level questions to the system in order to understand how a failure came up or simply understand how the system is working.

Information to Log

In order to understand which are the informations to log it is important to make some consideration. In a distributed environment:

- the entities which make up an application are active (processes or agents);
- the interactions are based on message exchange mechanism;
- the process life time is connected to the application life time; the life time of an agent are usually independent from life time of a specific application;
- the logical architecture can be set by different patterns: client-server, peer to peer, etc.
- the middleware realize the physical and logical connection between entities (subsystem, service, object, component, process, agent, etc.).

From an high level point of view the kind of applications like SKA TM define a logical network of interactions which it is composed by different nodes that interact each other and some of them act as coordinator or controller (at least for the online part of the system). So it is important to log:

- *Node[†] identification,*
- *Node signal[‡] declaration,*
- *Node interactions,*
- *Actions and loops.*

[*] Work supported by the Italian Ministry of University and Research (MIUR)

[†] In a network, a node is a connection point, either a redistribution point or an end point for data transmissions. In general, a node has programmed or engineered capability to recognize and process or forward transmissions to other nodes.

[‡] An information usable by a node in order to control the behaviour of another one or its behaviour in function with the one from another node.

[**] Doing something programmatically means that you can do it using source code, rather than via direct user interaction or a macro

Copyright © 2015 CC-BY-3.0 and by the respective authors

ISBN 978-3-95450-148-9

KNOWLEDGE-BASED SYSTEM APPROACH

A knowledge-based system is a computer program that reasons and uses a knowledge base to solve complex problems. It is composed by two types of sub-systems: a knowledge base (facts[§] about the world) and an inference engine (logical assertions and conditions about the world). To build it, it is possible to use a formal language like Prolog.

A *fact* must start with a *predicate* (which is an *atom*[**]) and end with a *fullstop*. The predicate may be followed by *one or more arguments (*separated by commas) which are enclosed by *parentheses*. The arguments can be atoms (in this case, these atoms are treated as constants), numbers, variables or lists. The formalism can be summarized in this way: *predicate(arg1, arg2, …, argN).*

Besides, the prolog language can be seen as a language for database queries; in fact, in a relational database a *tuple* is a generic element of a relation with attributes.

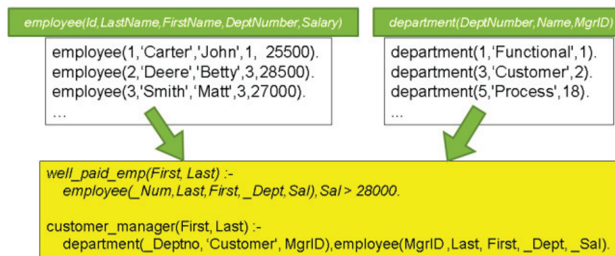

Figure 1: Prolog as database.

In Figure 1, it is shown how it is possible to represent two tables with the prolog language (the table *employee* and the table *department*) and two possible queries: the first one which give the names of all employees earning more than 28000 and the second one which give the name of the manager of the department Customer.

Interface

In order to allow software developers to use such a formalism without actually knowing it, a possibility is to look at the symbolic representation and try to generalize it following some general rule in order to finally get a *prolog theory*.

Based on the definition of a fact, a very simple way to log programmatically informations can be the following one:

logger.log(level, keyword-predicate, arguments);

where *level* is an enumerator, the *keyword-predicate* is a string and the *arguments* are an array of strings. This method can translate the informations into a Prolog fact, avoiding the developers to study and use the Prolog language. The only thing they have to do is indeed to choose a list of words representing the informations needed to store. So it is important to have a strong analysis based on UML or other modelling language. This is because the words used to log must have a sense and must not be chosen casually. Anyway, this is only a method to store informations and not a mechanism to query them. There is still a need to learn the Prolog language in order to gain full advantage from the use of the declarative approach (for simple query it is possible to make a generic tool, for example for level, date). Besides it will be always possible to search for a string directly in the text file just opening it with a notepad tool.

Compared to an old logging system (using for example the natural language), the proposed one aims to formalize every phrase in a sort of database (a Prolog database) where tables are dynamically built on the predicates chosen. Let's take the following log line:

Attaching appender named [CONSOLE-REQUEST] to Logger[TangoClientRequests]

The Prolog translation could be the following:

attaching(appender('CONSOLE-REQUEST'), logger('TangoClientRequests')).

Working in this way will give us the advantage to easily search for 'appender' or 'logger' (because we defined it as predicates) or directly searching for the predicate 'attaching'.

TANGO EXAMPLE

Tango[1] has been chosen by SKA Organization as a common middleware for communication and development. In this work it is used the same framework to show a proof of concept concerning a declarative approach based on the Prolog language.

A Counter Device

A counter device is a software that implements the following tango commands:

- *PlusOne*: increase the counter by one;
- *MinusOne*: decrease the counter by one;
- *Read*: read the value of the counter;
- *SetMaxValue*: set the max value for the counter (the counter will start from 0 and will never reach the maximum value but only the max – 1);
- *SetCounterConsumer*: set the name of the device which will use it (for logging purpose);
  Reset: set the counter value to 0.

The device will throw an event every time its value changes (a Tango change event) and every time the value is reset (when the value is equal to zero a tango user event

§ A fact is a truth about the real world that can be represented as a symbol, in order to be easily manipulated by programs.

** An atom, in Prolog, means a single data item like a string or a symbol like likes, john, and pizza, in likes(john, pizza).

ISBN 978-3-95450-148-9

Copyright © 2015 CC-BY-3.0 and by the respective authors

is thrown). Based on the analysis on the informations to log, a possible representation can be the one shown in Table 1.

Table 1: Log Information Expected from the Counter

Information	Prolog formalism	Counter example
Node identification	entity(entity-name, entity-type).	entity(counter, ‘LRU’).[††]
Node signal declaration	declares(entity-name, signal).	declares(counter, ‘+’) declares(counter, ‘-’) declares(counter, reset). declares(counter, read).
Node interactions	relation-word(emitter-name, receiver-name, signal, …).	interaction(consumer, counter, ‘+’).

Based on Table 1, it is possible to imagine a real log like the following:

entity(date(2015,5,15,10,50,0,0,-,-),counter, 'LRU').
declares(date(2015,5,15,10,50,0,0,-,-),counter, '+').
declares(date(2015,5,15,10,50,0,0,-,-),(counter, '-').

entity(date(2015,5,15,10,50,0,0,-,-),clock, 'LRU').

interaction(date(2015,5,15,10,50,0,10,-,-), clock, counter, '+').
…
interaction(date(2015,5,15,10,50,0,90,-,-), clock, counter, '+').
interaction(date(2015,5,15,10,50,0,100,-,-), clock, counter, '-').
interaction(date(2015,5,15,10,50,0,110,-,-), clock, counter, '-').
interaction(date(2015,5,15,10,50,0,130,-,-), clock, counter, '-').
interaction(date(2015,5,15,10,50,0,140,-,-), clock, counter, '+').
interaction(date(2015,5,15,10,50,0,160,-,-), clock, counter, '+').
interaction(date(2015,5,15,10,50,0,170,-,-), clock, counter, '+').
…
event(date(2015,5,15,10,51,0,0,-,-), counter, minute).

Figure 2: Real Log Example.

With this log file it is possible to query it for retrieving all the messages prior a certain date:
interaction(S,X,Y,Z),S@<date(2015,5,15,10,50,0,31,-,-).

But it is also possible to make complex query like counting ‘+ signal’ and ‘- signal’ to check if the difference is what we expect:
findall([], interaction(D,X,Y,'+'), L), length(L, N), findall([], interaction(D, X,Y,'-'), L1), length(L1, N1), Tot is N-N1.

[††] An LRU in SKA terminology means “ line replaceable unit”.

An Emom Device

In order to demonstrate the power of a declarative approach for the logging service, it is helpful to introduce an example that allows to generate a meaningful log.

Emom, in the context of a gym, means “every minute on the minute” and it is a technique for training for which a gymnast has to make an exercise every minute in less than a minute. Usually in an even minute is an exercise while in an odd minute is another one. To help people with this practice many app have been created (usually for smartphone) which indicate with a different colour whether the current minute is odd or even (see Figure 3: Emom Gui Even/Odd).

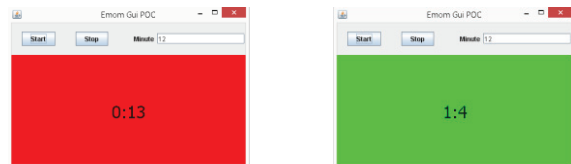

Figure 3: Emom Gui Even/Odd.

Based on the counter example, it is possible to build the Emom device with the help of two Counter Devices, one for the seconds and the other one for the minutes. In the Emom device, there will be a thread which will send a command every second to the seconds counter and, in case of the reset event, it will send a “PlusOne” command to the minute counter. When a minute changes an event is raised from the device: an “Odd” event if the minute is odd, an “Even” event if the minute is even.

The commands owned by the device are:

- *Start*: start the emom process;
- *Stop*: stop the emom process;
- *Reset*: reset the two counters;
- *SetMinuteCounterDevName*: set the name of the device for minute counting;
- *SetSecondCounterDevName*: set the name of the device for second counting.

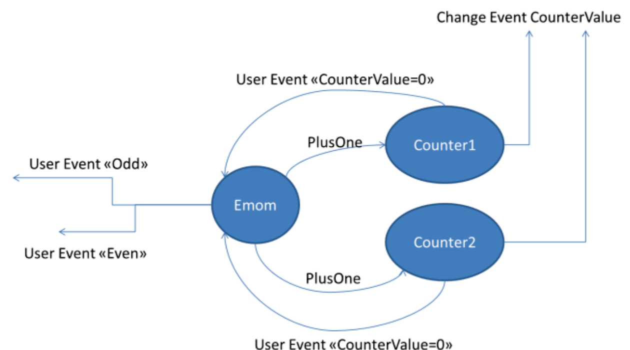

Figure 4: The Emom device.

Copyright © 2015 CC-BY-3.0 and by the respective authors

ISBN 978-3-95450-148-9

The emom process is summarized in Figure 4:

1. The emom device sends a «PlusOne» command every second to the first counter (for second);
2. An event is generated where the counter second reset its value;
3. When happens a «PlusOne» command to the second counter (for minute);
4. An event is generated where the counter minute reset its value;
5. If minute is «odd», the emom throw the corresponding event otherwise an «even» event.

Log4Prolog: A Tango LogViewer Branch

To be able to take advantage of the declarative approach with Prolog log message, it is necessary to modify the standard Tango LogViewer creating a software branch. The purpose is to create a plugin architecture so that a developer can create a specific filter for his development or simply use a logic console editor to ask some particular or specific question to the engine.

Every filter is a custom class that has to implement a specific interface called *IFilter*. When starting the application the application instantiates the filter present in the configuration file (an xml file) and adds the plugin in the application. A filter has the ability to show a new popup (with a title and size) or directly add a new control inside the main control panel of the user interface. The running application is shown in Figure 5: The Log4Prolog Application where it is shown six plugin: four of them are simple controls added to the main user interface (which emulate the standard filters of the old Tango LogViewer app) and the other two are define two dialogs. The first one is a prolog console where an expert can ask high level question while the second one (the upper one in the figure) is a specific plugin created for the emom example (called emom log analyser).

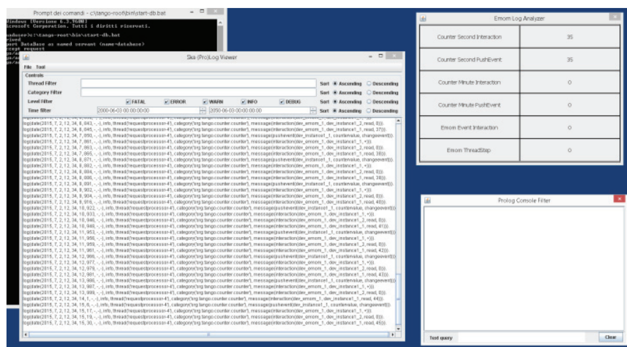

Figure 5: The Log4Prolog Application.

In specific the Emom log analyser define the logic query described in Table 2.

Table 2: Query Explanation

Title	Log query explanation
Counter second interaction	Find all interaction messages sent *to* the second counter device then count them.
Counter second pushEvent	Find all push-event[‡‡] messages sent *from* the second counter device then count them.
Counter minute interaction	Find all interaction messages sent *to* the minute counter device then count them.
Counter minute pushEvent	Find all push-event messages sent *from* the minute counter device then count them
Emom event interaction	Find all event-interaction messages sent *to* the emom device then count them.
Emom threadStep	Find all thread-step messages sent *from* the emom device then count them.

CONCLUSION

In this document a declarative approach in a logging system has been described. A good way to develop software is thinking in term of test and unit test. In addition to these techniques it is desirable to think in terms of logging so that the log file can be a way to read what the software is doing. If it is chosen to use a declarative language it is possible to work with files ready to be queried because of the formalism used.

The main disadvantage is related to the declarative language itself. However, by thinking the language as aimed at creating artificial intelligence or as a *sophisticated database language* (more NoSQL-like than most NoSQL approaches), it appears just like any other tool in the collection of a software architect that can help in solving specific problems.

REFERENCES

[1] JM. Chaize, A. Goetz, WD. Klotz, J. Meyer, M. Perez, E. Taurel, P. Verdier, "The ESRF TANGO control system status", ARXIV , November 2011

[‡‡] A push event in Tango occurs when the developer wants to force an event to be thrown.

ISBN 978-3-95450-148-9

Copyright © 2015 CC-BY-3.0 and by the respective authors