Monitoring and State Transparency of Distributed Systems

Martin J. Logan
mlogan@futuresource.com
Futuresource

Abstract

This paper presents the **System Status** suite of applications. These applications are used to provide a simple, uniform, and low developer cost system for exporting and tracking the state of OTP applications and services over a distributed server farm based network. The terms, simple, and low developer cost, will be elaborated on later in the paper. The system is intended to provide no formalized management framework. The **System Status** suite is specifically a state/status export and monitoring infrastructure.

Categories and Subject Descriptors

D.2.5 **[Testing and Debugging]** *Debugging aids, Diagnostics, Distributed debugging, Monitors*

General Terms

Management, Measurement

Keywords

Distributed computing, monitoring, server farm, services oriented network, concurrent programming

1. Introduction

To explain the motivation behind this software and its subsequent set of requirements the environment for which it is intended must be described. Small to medium sized businesses, those with under 100 employees, where development is focused on short development life cycle applications. I have been employed by this class of business for the life of my career. I have noticed that key decision makers often must subordinate monitoring and infrastructure to elements of development that have a higher perceived impact on the bottom line. Stated another way; key decision makers facing competition from larger companies as well as limited budgets and tight deadlines strive to achieve the highest **R**eturn **O**n **I**nvestment(ROI) possible by providing customer visible features ahead of infrastructure development. Applications need to be completed quickly and therefore the more complex monitoring solutions such as CMIP which require many development hours, lots of configuration to implement, and in many cases experienced personnel to be effective, are not suitable in this environment[7]. The **System Status** suite of applications was built to allow developers to add value through custom monitoring of their applications all within a very limited time frame.

The social constraints that the **System Status** suite faces are discussed first in this paper but this is not meant to indicate that there does not exist stringent technical requirements. Small to medium sized businesses typically run a server farm architecture. These networks are highly distributed collections of "inexpensive" machines with applications running on top of them that must be able to handle the frequent addition of new machines and applications as well as the removal of old ones. To further complicate things the trend seems to be moving towards a more service oriented architecture[1] wherein services are deployed on the network and a variety of physically separated applications then rely on these services. While applications and machines are added and removed the existing applications and services must continue providing a seamless experience to the customer. This is exciting from a business and quality of service perspective but provides an engineers attempting to create an effective monitoring tool set with some not so trivial hurdles to jump.

"We want to know when different OTP applications and resources enter and leave the network. We want to be notified of all events that our engineers consider important when they occur in our software on the network – oh and we want it to cost us nothing." The above pretty much summarizes the aim in developing the **System Status** suite**.**

This paper is structured as follows: First specific requirements required to meet the general constraints imposed by the business environment are discussed. Section 3 is a discussion of the specific technical requirements needed to exist in the server farm environment and cope with the distributed and dynamic nature of such an environment. Section 4 details the **System Status** suite implementation and the ways in which it satisfies all the requirements are made clear. Finally conclusions are discussed.

2. Business Environment Requirements

As explained in the introduction to this paper monitoring is often overlooked as being less critical in providing a solid ROI than the set of more customer visible features. It is unreasonable to assume that engineers are going to be able to talk key decision making personnel into believing otherwise in the majority of cases. Instead the direction that should be taken is to use a monitoring solution that is specifically designed to ease integration into software development projects within a limited time-frame. The focus must be on ease of use above all else. Information should be discovered and organized in an intuitive manner by the system. A further consideration from the business side is protection and utilization of existing investments by way integration with existing/legacy infrastructure. Engineers/developers, in the interest of building the perfect solution to a problem, are typically more than happy to scrap the old and bring in the new. This is not always cost effective and it will typically meet with strong opposition from those managing expenses. If there is an existing infrastructure **System Status** should make integration with that existing infrastructure possible and relatively painless. Core requirements stemming from business concerns are as follows:

1. *Minimal configuration on source side*, meaning under a few minutes spent on configuration per deployed application and preferably no configuration. The term *source* here is used in the same sense as it is in BSD Syslog. An application that emits information to be collected by other interested parties.
2. *Minimal training*, under a few hours, needed for maintenance and training personnel to begin to use collectors, including alert and inspection tools, effectively. Information in the system should be easily discovered by users. Information should be passed on to the user through intelligent organization of data. The term *collectors* is used in the same sense as it is with BSD Syslog to mean any application consuming messages emitted by source components.
3. *A sufficient set of preexisting collector class tools* in the suite to allow maintenance personnel to effectively view the state of source applications on the network with no added development.
4. *Integration into existing alert and monitoring facilities*.
5. *Simple easy to understand API* that allows developers to instrument code as simply as they would write log messages to a file.

3. Engineering Requirements

Some of the purely engineering requirements are generic to monitoring any networked environment while others are driven specifically by the rigors of the service oriented server farm environment.

1. *Event Notification* so that *c*ollectors are informed promptly of interesting events occurring on one of the sources.
2. *Selective filtering of events*. Events should have a structure that allows collectors to selectively match patterns so as to receive only pertinent data.
3. *Capability to detect new resources* when they enter the network. A resource in this context is defined to be a node or a process, the active entities in an Erlang system.
4. *Capability to detect the death of a resource.*
5. *The system must not overburden the network.*
6. *Monitor many clusters* irrespective of their cookies. Erlang networks are composed of clusters defined by net cookies. These clusters do not necessarily delineate monitoring domains. A monitoring system should be able to monitor many clusters at once.

4. Implementation

The **System Status** suite of applications consists primarily of **System Status**, its namesake, **Insight**, the **System Status Relay,** and **fs_message_bus.erl** module. Together these applications give the user a source a collector/inspector and some integration into a legacy system, namely BSD Syslog.

The **System Status** is built around the notion of events. These events indicate that that there has been an interesting occurrence in the system. Events are characterized by an id and a payload.

4.1 fs_message_bus.erl – transportation layer

This module provides the transportation for messages across the network over multicast udp. A user may publish over the bus as well as subscribe to messages picked up off of the bus. The **System Status, Insight,** and **System Status Relay** applications all rely on the **fs_message_bus** to send and receive messages.

4.2 System Status(the application itself, not the suite) – The source

The system status application is at the core of the **System Status** suite. This application serves as the source for all events. At the center of the system status system is its core event format. This is contained within the system_status.hrl file within the **System Status** application.

```
-record(system_status, {priority, now, node, application,
attributes, id, payload}).
```

Events are codified with the above record structure. The structure of these events is reminiscent of the BSD Syslog[3] message format. The event contains within it a priority integer, with meaning identical to that of BSD Syslog, a time stamp indicating the time the event transpired, the node that the event occurred on, the application that generated the event, some attributes indicating how to treat the event, a pattern matchable id, and finally the payload.

The system status application must be included in the release that contains applications that are to use it, this requires no configuration other than inclusion in the .rel file. There are four configuration parameters that can be altered but the defaults are intelligent and will suffice in the majority of cases. From the programmers perspective emitting events from an application is quite simple. This is by design. The programmer must supply an indication of the priority for all events. Priority designations would be different between an event indicating that some application is initialized and ready to go and an event serving as a critical notice that core data is corrupted. Priority is reflected in the actual functions called to create an event, such as critical/2 or info/3. The developer may supply a combination of only four attributes or simply not include any to employ a default event treatment. The developer then supplies the id and the payload. System status also supports other convenience functions. The functions count/2 and /3, for example, allow the developer to enable external awareness of the number of times a particular event has transpired.

When an event is sent into the System Status application it is immediately sent out over the fs_message_bus and labeled via its attribute list as being *causal*. The user also has the option to include the *refresh* attribute with an event. This causes system status to store the event locally and broadcast it out over the message bus periodically as a refresh. This is to ensure that a collector that arrived on the network late could get a snapshot of the state of things even though it missed that actual causal occurrences of particular events. Administrators have the option modulate the traffic produced by **System Status** by setting the priority level for broadcasting at start-up through configuration, at run time via the API, or through **Insight.** The way priority level effects the system is that if it were, for instance, to be set to *critical*(meaning level 2), then only events with a priority of 2 or lower will be emitted. Setting the priority level to *off*(-2) keeps the network totally clear of events if that is deemed necessary by administrative staff.

One of the stated goals of **System Status** is to organize monitoring information effectively. Those familiar with mnesia have typed the command mnesia:info(). This command delivers a picture of the state of the mnesia application at the time the call was made. IBM calls this “snapshot monitoring”[6] and uses it in the monitoring system for DB2. This snapshot, determined by the engineers, contains salient data pertaining to the current state of the system. This data can give an observer an instant picture of the health or function of the given system. System status provides an abstraction for snapshots that takes just a few minutes for a developer to work up for a particular application. This is called a *template*. A template is contained within the following record.:

```
-record(template,{app_name, app_description, app_vsn,
                  template_name, template_description, ids}).
```

An application developer simply places the call-back, *system_status_templates/0,* in the -behaviour(application) call-back module. This function returns *raw template definitions* which are collections of the system status id's, that the code for the given application has been instrumented with. These id's, grouped intelligently, provide the snapshot of an aspect of the applications state.

Example:

```
system_status_templates() ->
   %% A list containing a single raw template definition
   %% {template_name, template_description, ids}
   [
    {data_integrity,
    "an idea of the integrity of the processed data",
    [{count,      parse_errors},      parse_errors,      {count,
math_errors}, math_errors]
   ].
```

The above demonstrates how a developer would create a template to illustrate the quality of the data in an application by incorporating various events already programmed into that application into a single template. The list of id's can be explicit id's as above but may also contain patterns such as {count, '_'}. An application may have multiple templates associated with it. This is a powerful feature for adding and spreading information with a minimum of developer time and effort. The end users of the system status system can query for all the templates on a given node via the system status API and go about collecting data in accordance with them using **Insight**.

4.3 Insight – Collector and Inspector

The **Insight** system is a specialized shell that allows interested personnel to inspect one or many of the OTP releases on a network that incorporate **System Status.**

Two of **Insight**'s main focuses are the discovery and the organization of the discovered information.. These focuses are two of the factors making **Insight** effective in the business an technical environments that it is designed for. An effective collector must be able to search out information on the network and make the operator aware of what is out there in an organized fashion. The first level of discovery **Insight** focuses on is nodes on a network. The **System Status** application sends an automatic heartbeat for its local node. This heartbeat allows the insight application to "see" and track nodes on the network irrespective of their cookie based clusters. The second level **Insight** focuses on to gather information are the **System Status** templates. An **Insight** user may enter the following command from the **Insight** shell:

```
I.shell ~> templates all all
```

Entering this command instructs **Insight** to display all templates from all known nodes for all applicatons. The display indicates to the user the name of the template, a description of the template, and the id's and patterns that comprise the particular template. The display also contains information indicating from which nodes and applications the templates came from as well as providing descriptions of source applications. With this information an operator, having previously known nothing of the layout of the network, can go about taking real time snapshots of the applications of interest.

Insight provides two primary mechanisms for inspecting resources at the id/payload level. The first is a simple fetch. A list of all stored id's(remember many id's are stored for refreshes) on any combination of known nodes can be obtained through **Insight.** Once id's are known an operator can query any of the id's for its payload. This method is not entirely powerful by itself. The second method is to use a *view.* A view is a tool for displaying a real-time updating screen displaying the current state of any combination of node's, application's, template's, id's, and payload's. Basically a user tells the system to display a group of id patterns for a any combination of nodes or simply all the nodes known. This, in essence, allows a user to know and execute the following plan of action;

"I have just discovered that *application X* has a template called *errors.* This templates description indicates it will offer me information about why *Application X* is having difficulty. I will display a real time updating screen containing all id patterns contained within the *errors* template for all the nodes running *application X* on the network to aid in diagnosing the problem"

This can be accomplished through **Insight** with three words typed into its shell.

```
I.shell ~> screen all errors
```

The command above tells insight to display a screen view for all nodes and applications containing a template named errors. This command results in:

```
Insight(TM) - Screen View

Display: tmp
Nodes: all
Match Patterns: [math_errors, deaths, parse_errors, {count,
parse_errors}]
==================================
Key:  parse_errors

Node:  martin_time_series_rel@mecha.futuresource.com
Last Update:  16:36:9 on Jul 29, 2004   Time Created:
16:36:9 on Jul 29, 2004
Value: {parse_error, {packet, <<12,0,0,0>>}}

Node:  eric_time_series_rel@inferno.futuresource.com
Last Update:  16:35:3 on Jul 29, 2004   Time Created:
16:35:3 on Jul 29, 2004
Value: {parse_error, {packet, <<12,0,0,0>>}}

--------------------------

Key:  {count, parse_errors}

Node:  martin_time_series_rel@mecha.futuresource.com
```

```
Last Update:  16:36:20 on Jul 29, 2004   Time Created:
16:36:20 on Jul 29, 2004
Value: 42023

Node:  eric_time_series_rel@inferno.futuresource.com
Last Update:  16:35:15 on Jul 29, 2004   Time Created:
16:35:15 on Jul 29, 2004
Value: 40234
```

The data on the screen above refreshes as new data is received by **Insight**. The data can come from any node capable of emitting an event, over the message bus, that can reach this **Insight** instance. If a node were to go away the operator would see its data disappear. If a new node were to enter the system and emit the ids this view matches that data would be promptly displayed. From the data we can easily see why these applications might be having a problem. This information can be gathered with **Insight** by personnel with little to no training.

4.4 Relay – Legacy integration(BSD Syslog)

The system status relay is an example of converting the simple core **System Status** event message format into another protocol for compatibility with a legacy system. The system status relay sits on the network and listens to all traffic based on patterns indicated by its configuration(default is all traffic). The relay converts the captured **System Status** messages and converts them into BSD Syslog formatted messages. These converted messages can then forwarded to any BSD Syslog collectors on the network. The only salient pieces of configuration for the relay is the *priority level* which indicates the priority level events must meet or exceed if they are to be forwarded on and the BSD collectors. Little more can be said of this component.

5. Conclusions

Small to medium sized businesses face specific challenges that can make it difficult to implement effective monitoring for their software products. This paper has outlined requirements that must be met in order to practically add value for these businesses through monitoring.

1. *Minimal configuration on source side*
   System Status only has 4 configuration options and the defaults are intelligent.

2. *Minimal training*
   The facilities for information discovery and effective organization(node heartbeats, templates, views) and the demonstrated simplicity of the **Insight** command set meet this goal.

3. *A sufficient set of preexisting collector class tools*
   Insight has a fairly rich set of tools including its *views*. Support from interested developers would greatly enhance this area.

4. *Integration into existing alert and monitoring facilities*
   System Status Relay is a start. Much more can be done here but the start has been made.

5. *Simple easy to understand API*
   The **System Status** API is short, elegant, and intentional in nature.

6. *Event Notification*
   This lies at the heart of the **System Status** suite of applications. Events are formatted by the system_status record.

7. *Selective filtering of events*
   Yes. Pattern matching can be employed similar to that of ets:match functions.

8. *Monitor many clusters*
   System Status can inspect nodes regardless of network cookies.

9. *Capability to detect new resources*
   Insight leveraging the information broadcast by **System Status** means there is no need to statically configure anything having to do with the locations of resources to be inspected on the network.

10. *Capability to detect the death of a resource*
    Insight uses Erlang's native monitor_node/2 as well as the **System Status** node level heartbeats to track the comings and goings of a nodes. There is also a process level heartbeat that can be used to track process lifetimes.

11. *The system must not overburden the network*
    Administrators have the ability to set the priority level on any node from 7 to *off(-2)* thus giving them the option to modulate traffic all the way up to the point of completely preventing any network traffic caused by **System Status**.

The **System Status** suite of applications meets the initial requirements. Further development could focus on an message_bus architecture that allows for completely reliable communication, providing more built in statistical analysis tools, or providing finer grained control of stored events on a local **System Status** instance(reset, clear, append-payload). Other areas of weakness are the lack of tools for integration with other legacy systems and the need for a good visual alarm/alert system. That said, the **System Status** suite provides the target audience with added value while not incurring heavy costs and has demonstrated its usefulness in a production environment over the past six months.

6. References

[1] Sun Microsystems, *Java Management Extensions*, white-paper, http://java.sun.com/products/JavaManagement/wp/ online
[2] Mercury Interactive, *Agentless Monitoring: An Innovative Paradigm for Monitoring Mission Critical Systems,* www.mercuryinteractive.com online, 2003.
[3] RFC 3164 (rfc3164) - *The BSD Syslog Protocol*
[4] Mercury Interactive *Agentless Monitoring - Reducing the Total Cost Of Ownership (TCO) of System Monitoring,* www.mercuryinteractive.com online, January 1, 2004
[5] Enterprise Management Associates *Capturing and Managing the Details,* . http://www.statscout.com/papers.html Online, April, 2004
[6] IBM, *System Monitor Guide and Reference*, http://webdocs.caspur.it/ibm/web/udb-6.1/db2f0/db2f002.htm online
[7] Carnegie Mellon University ***Simple Network Management Protocol***, http://www.sei.cmu.edu/str/descriptions/snmp_body.html online, 2004