Gridifying IBM's Generic Log Adapter to Speed-up the Processing of Log Data

Claudi Paniagua
IBM GTS, Virtualization and Grid Computing EBO
SPGIT IMT IT, Barcelona, Spain
cpaniagua@es.ibm.com

Fatos Xhafa
Departament de Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya, Barcelona, Spain
fatos@lsi.upc.edu

Thanasis Daradoumis
Department of Computer Science, Multimedia, and Telecommunication
Open University of Catalonia, Rbla. Poblenou, 156. 08015 Barcelona, Spain
adaradoumis@uoc.edu

Abstract

Problem determination in today's computing environments consumes between 30 and 70% of an organization's IT resources and represents from one third to one half of their total cost of ownership. The first step to cutting down costs and to enable autonomic computing systems is to have all parts of the system report status in a common log data format and semantics. The Generic Log Adapter (GLA) is a generic parsing engine shipped with the IBM's Autonomic Computing Toolkit that has been conceived to convert proprietary log data into a standard log data event-based format in real time. However, in order to provide generic support for parsing the majority of today's unstructured log data formats the GLA makes heavy use of regular expressions that incur in performance limitations. Current approaches proposed to increase GLA's performance have revolved around fine-tuning the set of regular expressions used to configure the GLA for a particular log data format or writing specific parsing code. In this work we propose a very new approach consisting in transparently parallelizing the GLA by taking advantage of its internal architecture and the fact that structuring log data is a task that lends itself very well to parallelization. We present a master-worker strategy that "gridifies" the GLA efficiently in a completely transparent way for the user.

1. Introduction

The goal of problem management, as defined by the IT Infrastructure Library [1], the *de facto* global service management standard, is to minimize the impact of situations in the IT infrastructure that adversely affects the business and to prevent those situations by initiating actions to permanently correct their root cause. Problem management in today enterprise information systems is not an easy task: troubleshooting IT problems in medium and large companies can consume from 30 to 70% of the company's IT resources and outage costs per hour on business-critical information systems can range from thousands to millions of dollars [2].

One of the factors contributing to the difficulty of problem management is the multitude of different ways in which the different parts of an enterprise information system do report status. Log files are a common strategy for this, but even then a simple web-based business application may easily contain as many as 25 to 40 different log files, each one reporting status information using its own (often inconsistent) data format and semantics. Extracting out what's going on in the business application as a whole from these fragmented and inconsistently formatted data sources is a time-consuming and error-prone manual process that is only done reactively and off-line after a problem

Proceedings of the First International Conference
on Complex, Intelligent and Software Intensive Systems (CISIS'07)
0-7695-2823-6/07 $20.00 © 2007 IEEE

IEEE COMPUTER SOCIETY

has occurred in order to diagnose it. The disparity and lack of consistency in both the format and semantics of log data makes it more difficult to write management tools that ease problem determination; less, proactively monitoring and correlating this log data in real time in order to automatically identify problems as they happen (or even before they happen).

The goal of autonomic computing [3] is to provide open, intelligent, resilient systems with self-management characteristics. Though ambitious, there's an evolutionary roadmap to get to autonomic computing. The first step is to standardize log data format and semantics to enable the automation of problem management activities across the entire enterprise information system.

The Common Base Event (CBE) [4] is IBM's implementation of the WSDM Event Format (WEF) OASIS standard [5]. CBE is an XML based universal log data format defined in XML Schema that organizes log data in events. An event is defined as the occurrence of a situation of interest. Log data sources are supposed to report status information as a temporal succession of discrete events, i.e., occurring situations. In CBE each situation is represented as a "3-tuple" structured XML document: (1) the component originating the situation, (2) the component observing the situation, and (3) the data that describes the situation, including correlation information. An Event Driven Architecture (EDA) [6] allows connecting event emitters to event consumers in real time without introducing any coupling between them and is well suited for supporting powerful techniques for monitoring and problem management such as complex event processing [7]. The Common Event Infrastructure (CEI) [8] is IBM's implementation of the main building blocks of an EDA and a fundamental piece of IBM's autonomic computing architecture that mediates between the CBE emitters and the problem management and monitoring tools.

Because there is no cost-effective way to change existing products and legacy applications to log data in the CBE format, the IBM autonomic computing architecture includes adapters to translate disparate existing logs to the CBE format. The IBM's Generic Log Adapter (GLA) [9] is an implementation of such an adapter conceived to ease the transformation of existing log data to the CBE format in real time. We will call *log data normalization* the process of transforming existing log data to the CBE format.

In this paper we are concerned with the efficiency of processing log data introduced above. Indeed, the computational cost is the main obstacle to processing this data in real time [3] as it is very costly and due to this in real situations this processing tends to be done offline in order to avoid harming the performance of the logging application. Certainly, sequential approaches for the processing of log data cannot overcome this problem due to the huge amount of data to be processed. Grid technology [10] is increasingly being used to reduce the overall, censored time in processing data. Computational Grids are thus an attracting alternative for the problem of processing in real time or in quasi real time large amounts of log data collected during the daily activity of IT enterprises.

By considering a Grid-based approach for processing log data, we show the benefits of the Grid by offloading the online processing of log data onto the grid. Moreover, we show how a simple Master-Worker scheme sufficed to achieve considerable speed-up. We notice that our approach is generic and can be applied for structuring event log data in general.

The rest of the paper is organized as follows. In Section 2 we explain the normalization of Log Data with IBM's Generic Log Adapter. Section 3 presents some considerations about the performance of the Generic Log Adapter. Section 4 introduces the Master-Worker parallel approach for structuring and processing log data and some details are given in Section 5. In section 6 we present the most relevant results of this work.

2. Normalizing Log Data with IBM's Generic Log Adapter

This section describes the GLA architecture and how it processes log data sources to generate and output CBE instances [11]. The GLA is written in Java and is architected following a *chain of responsibility* design pattern [12] that chains five different types of components corresponding to the five different phases in which the GLA organizes the normalization of log data (see Fig. 1). These component types are in the order in which they are arranged in the chain: (i) the *sensor* component: this component monitors one log data source (i.e. a log file) reading it line by line as it changes. When the sensor has read a preconfigured number of new lines it passes them to the extractor component; (ii) the *extractor* component: this component receives a collection of lines from the sensor component and parses it to delimit the log record boundaries; (iii) the *parser* component: this component receives a collection of log records from the extractor component and parses them to map each one to a set of CBE attributes; (iv) the *formatter* component: this component receives a collection of sets of CBE attributes from the parser component and for each one builds the corresponding CBE instance based on the attributes of the set, and (v) the *outputter* component: this component receives a collection of

Proceedings of the First International Conference
on Complex, Intelligent and Software Intensive Systems (CISIS'07)
0-7695-2823-6/07 $20.00 © 2007 IEEE
IEEE COMPUTER SOCIETY

CBE instances from the formatter and persists or sends them to somewhere else in the infrastructure.

At runtime a component is an instance of a Java class. GLA components implement the *IComponent* interface that defines methods for managing the component properties and for starting and stopping the component. The *IComponent* interface is furtherly extended by two additional interfaces, *IContext* and *IProcessUnit*. The *IProcessUnit* defines the handler method of the chain, *processEventItems*, which is implemented by each component in the chain.

The chain is managed and orchestrated by a *context* component, which implements the *IContext* interface. The remaining interfaces, *ISensor*, *IExtractor*, *IParser*, *IFormatter* and *IOutputter* extends the *IProcessUnit* interface to provide specific methods for each one of the *sensor*, *extractor*, *parser*, *formatter* and *outputter* components respectively (see Fig. 2).

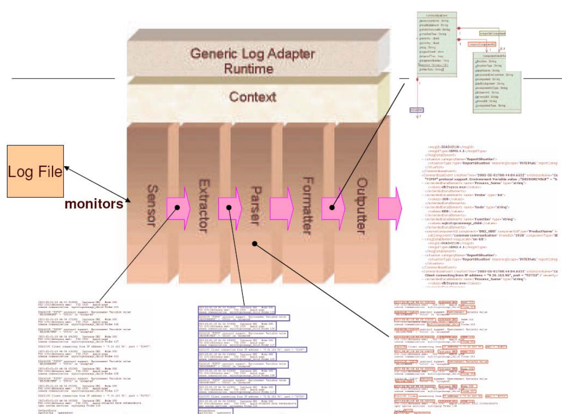

Figure 1: Architecture of the GLA

The GLA provides default implementations for all these interfaces but it is also architected following a plug-in design that allows the user to plug custom developed components. In fact, the specific java classes that conform a context (a chain of components) to normalize a particular log data source, together with their configuration parameters, can be specified using an XML file, called the *adapter* configuration file. The GLA takes this configuration file as an initial argument and instantiates the chain as configured. One can define more than one context in the same *adapter* file, thus the same GLA instance is able to normalize more than one different log data source. The GLA ships with Eclipse [13] that allows to visually configuring contexts, as well as to test and debug those contexts on sample log data (see [14]). The output of the development environment is an XML file that can be used to instantiate a GLA instance that will transform the log data sources as described by the XML file.

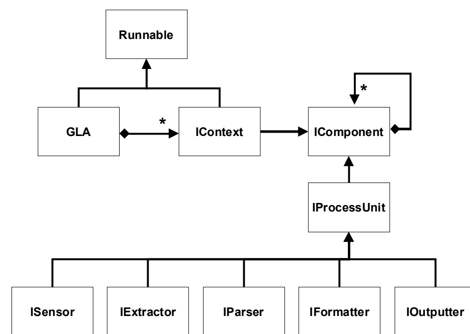

Figure 2: Diagram of GLA's Class Hierarchy

3. Considerations of the Performance of the Generic Log Adapter

In this section we analyze some performance considerations of the GLA that motivated our parallel approach. We start by taking a slightly more formal look at the process of normalizing log data. Log data normalization can be modeled using elements of formal language [15]. The log data generated by a log data source between two instants in time can be represented by a word (i.e. a string), **,** from a given alphabet, , that contains all the characters that the log data source may possibly use to represent log data. The *sensor* component then reads this word as it is being generated, thus outputs a sequence of sub-words of , say, $_1$, $_2$,…, $_m$. The extractor component acts on these sub-words one at a time, outputting a collection of sub-words, $\boldsymbol{E}(\ _i) =\ _{i1},\ldots,\ _{in}$, of $_i$ each one corresponding to a different log record or message and thus verifying one simple but very important property: they are independent units of structure, i.e., each sub-word contains all the information the parser component needs to access in order to be able to map it into a set of CBE attributes, $\boldsymbol{P}(\ _{ij})$, that the formatter component will transform into a CBE instance, $\boldsymbol{F}(\boldsymbol{P}(\ _{ij}))$.

Now we can see that to normalize a piece of log data ω_i, we need to compute $\boldsymbol{F(P(E(\omega_i))}$ where $\boldsymbol{P(E(\omega_i))=P(\omega_{i1})\ldots P(\omega_{in})}$ and $\boldsymbol{F(P(E(\omega_i))=F(P(\omega_{i1}))\ldots F(P(\omega_{in}))}$. Let's roughly compare the relative time complexity of the computations $\boldsymbol{E}$ and $\boldsymbol{P}$ for the case of the default implementations for the extractor and parser components that come with the GLA[1]. Both implementations use regular expressions specified by the user at configuration time through the Eclipse-based tooling, however, the way in which the two

[1] The computation $\boldsymbol{F}$ that the formatter component performs is essentially different from $\boldsymbol{P}$ and $\boldsymbol{E}$ and thus cannot be compared. It boils down to creating *n* CBE instances, $\boldsymbol{e_j}$, and then filling it as specified by $\boldsymbol{P(\omega_{ij})}$.

Proceedings of the First International Conference
on Complex, Intelligent and Software Intensive Systems (CISIS'07)
0-7695-2823-6/07 $20.00 © 2007 IEEE

IEEE COMPUTER SOCIETY

types of components use the regular expressions differ considerably and directly impact in performance. The extractor's default implementation uses two regular expressions, one to define the pattern that starts a new log record and another one to define the pattern that ends a log record. The extractor scans ω_i looking for these patterns, each time it finds a match for the start pattern it includes the characters that follow into a new sub-word ω_{ij} until it finds a match for the end pattern[2].

On the other hand the sensor component default implementation uses an ordered collection of regular expressions for each CBE attribute that is to be filled from log data. It works as follows, for each sub-word ω_{ij}, for each CBE attribute to be filled and for each regular expression associated to the CBE attribute (in the order they were defined by the user) the sensor component scans ω_{ij} looking for a match, if one is found the matching characters are used as the value for the CBE attribute and no more regular expressions associated to this CBE attribute are essayed for ω_{ij}. If no match is found the CBE attribute if left with an undefined value. The time complexity of matching a regular expression in a string is directly proportional to (1) the length of the regular expression, (2) the complexity of the regular expression and (3) the length of the string. While the length of the string that the extractor and parser implementations need to scan is the same the extractor implementation only needs to essay at most two regular expressions, while the sensor component needs to essay usually a large number of regular expressions that tend to be complex and lengthy [16]. This has serious implications for performance; writing efficient regular expressions is thus important for the GLA [16,17,18].

Since the GLA's main processing loop is a chain of synchronous calls that must all finish before the next iteration can start, the parser component becomes a bottleneck: this may not be a problem if log data is generated at a slower rate than that at which the GLA is able to process it, however if this is not the case a remnant of log data pending to be processed is produced introducing thus a delay that may even increase over time and might eventually defeat the objective of being able to normalize log data in real time. On the other hand, even in scenarios where the GLA is able to process the aggregated log data generation rate in time, it might not be acceptable for the GLA to "steal" the CPU and memory resources required from production applications. It should be noticed that in today enterprise information systems log data is often tuned to be generated at slow rates for performance reasons, often leaving unlogged crucial information for problem determination. Being able to process more log data efficiently would allow to increase the amount of information logged thus easing problem determination. On the other hand when considering log data generation rates we should consider the aggregated rate of all log data sources running in the same machine which might considerably higher than that of a single log data source.

4. A Master-Worker Strategy to Parallelize IBM's Generic Log Adapter

Motivated by the previous considerations on performance we present a high level approach to parallelize the GLA using the Master-Worker (MW) paradigm [19,20] at the interface between the *extractor* and the *parser* components. MW has been widely used for developing parallel applications. In this model there are two different types of entities: *master* and *worker*. The *master* is in charge of the main flow of the program; it decomposes the main task into subtasks and sends these to the *worker*s, which process them and send back the result to the *master*, which uses them in its main flow of computation.

The MW model has proved to be efficient in developing parallel applications with different degrees of parallel granularity and is particularly useful when the partitioning of the problem is easy to compute and the dependencies between tasks are low or inexistent. As can be seen from the description of GLA from Sections 2 and 3, this is precisely the case for the GLA since: (i) the extractor component outputs independent units of structure which means that if the problem is partitioned using the boundaries of these units no dependencies between tasks will exist, and (ii) the input of the problem can be easily partitioned in these units of structure since, as we have seen, these can be done using at most two simple regular expressions. Given all the above, the GLA can be naturally parallelized using the MW paradigm by grouping the *sensor* and *extractor* components at the *master* side and leaving the *parser*, *formatter* and *outputter* components at the workers side (see Fig. 4) The advantage of using MW approach is threefold. First, we decouple the sensor and extractor from the parser, formatter and outputter components, that is, in our approach the master's main processing loop does not need to wait for the parser, formatter and outputter to finish its processing, the extractor component just passes the collection of log records to some worker and returns immediately. The worker then performs the rest of the processing asynchronously in another machine.

This effectively shortens the main processing loop at the master side, where the log data is being generated, to just the sensor and extractor components,

[2] One can specify whether the characters that match the start and end patterns should be included in the sub-word or not.

thus increasing the rate at which log data sources are monitored for changes. On the other hand we are able to offload the bulk of log data normalization computation to machines other than the ones that are producing the log data, which usually require as much resources as possible for their production running applications. Last but not least, we are able to normalize log data in parallel thus speeding-up the processing making it more real-time. However, there's a drawback in this approach: the master is not in full control of the size of the tasks that it sends to workers since log records can have arbitrary size and we do not control neither the rates at which log data sources do produce log data. In general we will be only able to play with the task size if the real time processing requirements are not very strict thus allowing us to accumulate log data of low volume log data sources until the task of the size is "big enough". We can conclude that a parallel implementation of log data normalization is applicable to high volume log data sources, but also to low volume ones provided that they have low real time processing requirements.

5. Transparent Parallelization of IBM's Generic Log Adapter

In order to experimentally test the feasibility of the MW paradigm for parallelizing the normalization of log data we have implemented a minimal Grid prototype that parallelizes the GLA. We used the Globus Toolkit 3.2 and we deployed the prototype on the Planetlab platform. The Globus Toolkit (GT) [21] is the actual *defacto* Grid middleware standard. Version 3 of GT (GT3) is a refactoring of version 2 in which every functionality is exposed to the world via a Grid service. Grid services are basically stateful web services. The core of the GT is a Grid service container implemented in Java that leverages and extends the Apache's AXIS web services engine. Planetlab [22] is an open platform for developing, deploying and accessing planetary-scale services. It is, at the time of this writing, composed up of 726 nodes hosted in 354 different sites. Each Planetlab node is an IA32 machine that must comply with minimum hardware requirements (i.e. 1GHz PIII + 1Gb RAM) running the same base software, basically a modified Linux operating system offering services to create virtual isolated partitions in the node, called *slivers*, which look to users as the real machine. Planetlab allows users o dynamically create up to one sliver in every node; the set of slivers assigned to a user form a *slice*; Planetlab nodescan run up to 100 concurrent slivers.

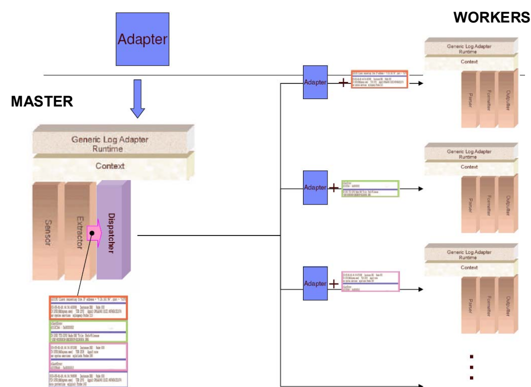

Figure 3: The scheme of MW paradigm applied to structuring log data

Moreover, our objective was to transparently parallelize the GLA, that is, to allow users to run their adapter files unmodified on the parallelized GLA (PGLA). In order to achieve this we reused the GLA code to make two versions of it: the master GLA and the worker GLA. The worker GLA is a grid service that exposes a single operation akin to the original GLA handler operation *processItemEvents* but that also receives the adapter file as an additional argument to the array of log records to be processed. The implementation of this operation uses the original GLA code to instantiate the chain of components as specified by the adapter file, then a minor modification is introduced that allows the chain to be initiated in its own thread at the parser component, bypassing the sensor and extractor components. This is the only modification required to the original GLA code to implement the worker GLA. In other words, the worker GLA executes exactly the same java bytecode (except at initialization time) to process the log data as the original GLA. This makes very easy and consistent the performance comparison between the sequential and parallel approaches. We deployed the worker GLA grid service on the GT3's containers of every sliver of our Planetlab slice.

On the other hand, the master GLA is again the original GLA code with a minor modification that forces the instantiation in the chain of a proxy component in between the extractor and parser components and modifies the chain execution so that only the third first components are called, that is, the sensor, extractor and proxy components. The proxy component reads in its *processItemEvents* from a configuration file the available GLA worker services method and implements a simple list scheduling strategy to forwards calls to the worker GLAs by

IEEE
COMPUTER
SOCIETY

invoking the corresponding grid services. This is a very simple scheduling strategy but notice that our objective was not to create a full-blown GT3-based MW implementation of the GLA but rather to show the feasibility of a transparent parallel Grid-based implementation of the GLA using the MW paradigm minimizing the amount of code to be modified from the original GLA.

6. Conclusions

In this paper, we first have motivated the need to structure and process in real time the large amount of information generated in IT enterprises. The problem of structuring and processing log data is gaining importance due its usefulness in problem determination, which is shown to be very costly and needing time superior to that of a single computer or of LAN of computers. We have considered the case of IBM's Generic Log Adapter and shown how to use a grid-based approach to efficiently speed-up the processing of log data. Although we have particularized our approach for the IBM's Generic Log Adapter, our approach is applicable in general to the structuring and processing of log data.

Thus, our results show the feasibility of parallelizing the problem of structuring any plain text event log data, achieving considerable speed up, provided that (1) the normalization algorithm's running time function, $f(n)$, be of strictly upper order than the transmission time function, n/B, that measures the time required to transmit a piece of data of size n for a bandwidth B. (i.e. $f(n) = \omega(n/B)$), and (2) the log data can be easily parsed (i.e. with few and simple regular expressions) in order to be broken in independent units of structure (i.e. log records). These conditions are expected to be satisfied by both log data and structuring algorithms, especially the ones that can be found in generic log data structuring/normalizing frameworks such as the GLA which are implemented using regular expressions.

Acknowledgments

This work has been partially supported by the Spanish MCYT project TSI2005-08225-C07-05.

7. References

1. IT Infrastructure Library, http://www.itil.org
2. B. Topol, D. Ogle, D. Pierson, J. Thoensen, J. Sweitzer, M. Chow, M.A. Hoffmann, P. Durham, R. Telford, S. Sheth, Th. Studwell, Autonomic problem determination : A first step towards self-healing computing systems, IBM, 2003
3. H.A. Müller, L. O'Brien, M. Klein, B. Wood, DTIC, Autonomic Computing, 2006
4. D. Ogle, H. Kreger, A. Salahshour, J. Cornpropst, E. Labadie, M. Chessell, B. Horn, J. Gerken, J. Schoech, M. Wamboldt, Canonical Situation Data Format: The Common Base Event V1.0.1, *IBM Corporation* 2003
5. Web Services Distributed Management**,** *OASIS Standard,* 2006
6. H. Gregor, Programming wthout a Call Stack – Event-Diven Architectures**,** http://www.eaipatterns.com/docs/EDA.pdf (2006)
7. L. David, C.F. Brian, Complex Event Processing in Distributed Systems**,** 1998
8. Best Practices for the Common Base Event and Common Event Infrastructure**,** IBM, 2006
9. Problem Determination Using Self-Managing Autonomic Technology, *IBM Redbook,* 2005
10. I. Foster, C. Kesselman. The Grid Blueprint for a New Computing Infrastructure. Morgan, 1998
11. E. Giguere, Create GLA components using Release 2 of the Autonomic Computing Toolkit, *IBM Corporation,* 2004
12. E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns, *Addison-Wesley* (2000)
13. Eclipse, http://www.eclipse.org/
14. An introduction to the Generic Log Adapter http://dev.eclipse.org/viewcvs/indextools.cgi/hyades-home/
15. J.E. Hopcroft, R. Motwani, J. D. Ullman, Introduction to automata theory, languages, and computation, Addison-Wesley, 2001.
16. B. Subramania. Improve the run-time performance of the GLA: A guide to writing efficient rule sets, http://www-128.ibm.com/developerworks/autonomic/library/
17. B. Subramania. Improve the run-time performance of the GLA, Part 2: A guide to writing efficient custom plug-ins, http://www-128.ibm.com/developerworks/autonomic/library/
18. R. Shetty, High-performance rule writing for the GLA. http://www-128.ibm.com/developerworks/autonomic/library/ac-glaperf/
19. J.P. Goux, S. Kulkarni, J. Linderoth, and M. Yoder. (2000): An enabling framework for master-worker applications on the computational Grid. In 9th IEEE Int. Symposium on HPDC, 2000.
20. E. Heymann, M.A. Senar, E. Luque, M. Livny (2000) Adaptive Scheduling for Master-Worker Applications on the Computational Grid. Int. Workshop on Grid Computing. LNCS, 1971, 214 - 227
21. The Globus Toolkit, http://www.globus.org/toolkit/
22. Planetlab, http://www.planet-lab.org/

IEEE COMPUTER SOCIETY