High-performance Linux cluster monitoring using Java

Curtis Smith and David Henry
Linux NetworX, Inc.
USA

Abstract

Monitoring is at the heart of cluster management. Instrumentation data is used to schedule tasks, load-balance devices and services, notify administrators of hardware and software failures, and generally monitor the health and usage of a system. The information used to perform these operations must be gathered from the cluster without impacting performance. This paper discusses some of the performance barriers to efficient high-performance monitoring, and presents an optimized technique to gather monitored data using the /proc file system and Java.

1 Introduction

Java technology has much to offer to the developers of cluster management solutions. Java is dynamic, flexible, and portable. These unique features make it an ideal foundation for building cluster management solutions over heterogeneous networks and platforms.

Java has an extensive library of routines for coping easily with IP protocols such as TCP and UDP, and for network programming on multi-homed hosts. This makes creating network connections much easier than in C or C++. Through the Java Native Interface (JNI), Java code that runs within a Java Virtual Machine (JVM) can interoperate with applications and libraries written in other languages, such as C, C++, and assembly.

Java is already a language of choice when building cluster monitoring or management systems. However the language is usually used for the front end or cluster host portion of the system, while a daemon written in C is installed on the cluster nodes [1] [10]. Despite the benefits provided by the Java programming language, the question remains: is Java efficient enough to replace the C daemon running on each node for high-performance cluster monitoring?

2 High Performance Monitoring

Tools for monitoring Linux clusters have traditionally provided a limited amount of data with delivery frequencies measured in seconds. High performance cluster monitoring is defined as the ability to efficiently gather data from nodes at intra-second frequencies. When dealing with large clusters, inefficiencies in the monitoring software become increasingly problematic. This is because running applications must coordinate with each other or with shared global resources. Interference on one node can affect jobs running on other nodes. An example of this is an MPI job that needs to synchronize all participating nodes.

One solution to this problem is to gather less data and transmit it less frequently. However if the objective is high-performance monitoring, this solution is unacceptable. A cluster that is heavily utilized should be monitored constantly and frequently. Local job schedulers must be able to make decisions quickly based on resource usage. Administrators often want to be notified of critical events and view historic trend data, which is unavailable unless the cluster is monitored constantly and consistently. Therefore utilizing more efficient algorithms, increasing transmission parallelism, increasing the efficiency of the transmission protocol and data format, and reducing redundancy are needed.

The performance numbers given in this paper are from code that runs the monitor at 100% CPU utilization. Therefore higher numbers indicate higher efficiency of the monitoring algorithm. Typically only those researching monitoring algorithms are interested in exclusively running monitoring code on a cluster. For all others, the monitoring rate must be throttled back to a reasonable level. For most cluster administrators, monitoring dynamic data every few seconds is adequate. So what are the advantages to monitoring at higher frequencies?

Profiling applications under development or tracking resource usage during execution can be useful for debugging or optimizing. The usage of dynamic resources such as memory, network, and CPUs can change very quickly for a given application. Being able to watch how an application utilizes these resources as it starts or as it runs is possibly one of the most interesting uses of high frequency monitoring.

Even if the user is uninterested in monitoring at high frequencies, if an algorithm is efficient it will consume fewer resources no matter what the monitoring frequency. This efficiency is more important in a heterogeneous cluster, where user jobs may be spread among faster and slower nodes. The slower nodes need the entire CPU to keep up and synchronize with faster nodes. A monitoring daemon taking CPU time on the slow node is on the job's critical path.

3 Monitoring Stages

Cluster monitoring primarily consumes two important resources: CPU cycles and network bandwidth. However, the resource consumption problem is fundamentally different for these two resources. The CPU usage problem is completely localized to the node, and is addressed by creating efficient gathering and consolidating algorithms. Network bandwidth however is a shared resource and is a problem of scale. The network bandwidth usage problem is addressed by finding ways to minimize the amount of data transmitted over the network.

To address these two issues, we divide cluster monitoring into three stages: gathering, consolidation, and transmission. The gathering stage is responsible for loading the data from the operating system, parsing the values, and storing the data in memory. The consolidation stage is responsible for bringing the data from multiple sources together, determining if values have changed, and filtering. The transmission stage is responsible for the compression and transmission of the data. This paper is concerned with optimizing the gathering stage of Linux cluster monitoring. The other stages are equally important to efficient monitoring and offer as many opportunities for optimization, but will only be discussed briefly in this document.

3.1 Gathering Stage

In Linux there are several ways to gather system statistics, each with advantages and disadvantages:

- Use existing tools: Standard and nonstandard tools exist which can perform one or more of the gathering, consolidation, and transmission stages, such as rstatd or SNMP tools. However the standard rstat daemon provides limited information and is slow and inefficient [2].
- Kernel patch/module: Several system-monitoring projects rely on the use of kernel modules or patches to provide access to monitored data [3]. In general, this is a very efficient way to gather system data. A problem with this approach however is keeping the code consistent with other changes in the main kernel source. A patch may conflict with other kernel patches the user wishes to use. In addition, the user must obtain and apply the patch or module before they can use the monitoring system.
- The /proc virtual file system: The /proc virtual file system is generally a fast and efficient way to perform system monitoring. The main drawback to using /proc however is keeping the parsing code synchronized with /proc file format changes. History has shown that the Linux kernel changes more frequently than the /proc file formats, so this should be less of a problem than using kernel modules or patches.
- Hybrids: Some monitoring systems implement a hybrid system, using a kernel patch/module for gathering the data and the /proc virtual file system as the interface to that data. The ideas presented in this paper apply in this case, since the data must be gathered from /proc using the same techniques.

3.2 Consolidation Stage

The responsibilities of the consolidation stage can be performed on the node, a cluster management host, or split between the two. In the interest of efficiency however, we almost exclusively consolidate on the node. The reason for this is that the node is the gatherer and provider of the monitored data. Two or more simultaneous requests for data should not result in two calls to the operating system to gather the data. Instead the data from the first request should be cached and provided to the second. This approach reduces the burden on the operating system and increases the responsiveness of the monitoring system.

The consolidation stage is also used to combine data from multiple data sources and at independent gathering rates. This is necessary because not all data can be

gathered at the same degree of efficiency, and because not all data changes at the same rate, or needs to be gathered at the same rate.

Another reason for consolidating at the node level is to reduce the amount of information included in the transmission. Many of the /proc files contain both dynamic and static data. By eliminating values that haven't changed since the last transmission, the amount of data sent by the node can be reduced substantially. Consolidation not only eliminates the transmission of dynamic values that change infrequently, but also addresses static values that never change.

3.3 Transmission Stage

Monitored data is almost always organized into a hierarchy. This is true of many of the files in the /proc file system. An effective means of encoding the hierarchical data into a form that can be transmitted efficiently is a responsibility of the transmission stage. The Java property file format is an effective and efficient means of storing hierarchical data, and can be easy produced with the provided Java APIs. S-Expressions have been suggested as another efficient way of transmitting this data [2].

A common point of discussion about transmitting monitored data is whether the data should be encoded in binary or text form. The issue usually is that binary data is more compact and is therefore more efficiently transmitted. However when using the /proc file system, monitored data is usually stored in human-readable form. Converting the data to binary form prior to transmission requires more processing resources and time. By leaving the gathered data as text, the node resources can be applied more to non-monitoring related work.

Leaving the data in text form provides these additional benefits:

- Platform independence: When monitoring in heterogeneous cluster configurations the byte order of the data is not always the same between machines. The use of a textual format solves this problem in a language and architectural independent way without imposing further processing requirements.
- Human readable formats: Textual data can be organized into formats that are humanly readable. If desired, this feature can ease the process of debugging or allow users to watch the data stream.
- Efficient compression: The text representation of numbers is comprised of characters from a set of 10 bytes as opposed to the set of 256 used in binary. The relative frequency of the digits and the patterns that they produce, allows dictionary and entropy based compression algorithms to be applied effectively.

4 The /proc Virtual File System

The /proc virtual file system (also referred to as procfs) is the Linux implementation of a virtual file system used by several UNIX operating systems, including Sun's Solaris, Linux, and *BSD [4][5]. It appears as a normal file system beginning at /proc and contains files having the same names as the process IDs of currently running processes. However files in /proc do not occupy disk space - they exist in working memory. The original purpose of /proc was to allow easy access to information about processes, but in Linux it is now used by every part of the kernel that has something to report.

Of the hundreds or even thousands of values provided by the /proc file system, we will focus on a minimal set necessary for cluster monitoring, this includes:

- /proc/loadavg: containing system load averages,
- /proc/memory: containing memory management statistics,
- /proc/net/dev: containing network interface card metrics,
- /proc/stat: containing kernel statistics,
- /proc/uptime: containing total system uptime and idle time.

A complete list of values available in these files is provided in Table 1. As would be expected the number of values provided by each file is different. In addition the file formats and time to gather the supplied information from the kernel are also different as will be seen later.

An important note about the /proc virtual file system is that each time a /proc file is read, a handler function is called by the kernel or owning module to generate the data. The data is generated on the fly, and the entire file is reconstructed whether a single character or large block is read. This is a critical point for efficiency, as any system monitors using /proc should gulp the whole file instead of nibbling at it.

Table 1. Monitor values.

File Name	Value
/proc/loadavg	1 minute load average 5 minute load average 15 minute load average total jobs total jobs running
/proc/meminfo	active memory inactive memory buffered memory cached memory total free memory total high memory free high memory total low memory free low memory shared memory swap memory swap cached memory swap free memory total memory
/proc/net/dev	*for each network interface card:* received bytes received bytes compressed receiving errors total receiving dropped errors receiving fifo errors receiving frame errors received multicast bytes total packets received

	transmitted bytes transmitted bytes compressed transmission errors total transmission carrier errors transmission collision errors transmission dropped errors transmission fifo errors total packets transmitted
/proc/stat	boot time number of context switches total number of interrupts total number of pages paged in total number of pages paged out total number of process total number of swaps in total number of swaps out aggregate cpu idle time aggregate cpu nice time aggregate cpu system time aggregate cpu user time *for each CPU:* individual cpu idle time individual cpu nice time individual cpu system time individual cpu user time *for each disk device:* individual disk block reads individual disk block writes individual disk io total individual disk io reads individual disk io writes
/proc/uptime	total system uptime total system idle time

The most critical inefficiency in monitoring /proc in the past has been reading /proc/meminfo. In order for Linux kernels earlier than 2.4.7 to gather memory statistics, the entire page structure and every block of swap had to be scanned. [2] This was a substantial problem that was fixed by version 2.4.14.

5 Implementation

So how do we go about writing a high-performance monitor in Java? Just as with any programming language there are efficient and inefficient ways to implement any algorithm. It is not always obvious which APIs will render the most efficient implementation.

Java provides a rich set of file IO classes. Included are stream based classes, block device based classes, and the new IO library provided in J2SDK 1.4.

Experimentation shows that in general the RandomAccessFile class is the best performing of the IO classes for basic block reads and writes to files. The newer 1.4 memory mapped classes do not provide improved performance in this case since the /proc virtual file system files are already memory mapped.

We begin with a basic implementation that at first glance appears quite reasonable. Knowing that the RandomAccessFile class is the most efficient, we base our code on it. For each load operation, the file is opened, parsed on a line-by-line basis, and then closed. We rely on the already implemented and optimized functions for string loading and manipulation provided by the Java language.

Our test's main function is designed to simply count the number of times a particular file (in this case /proc/meminfo) can be loaded, parsed, store its values in memory, and time the result. The tests were performed on a 1GHz Pentium III with 1GB memory and 99.8% idle CPU using the 2.4.18 version of the Linux kernel. The code for our first implementation is listed in Figure 1.

Figure 1. Code for parsing the /proc/meminfo file.

```
import java.io.*;
import java.util.*;

public class memoryfile
{
    private static final int TIME = 5000;

    private static RandomAccessFile mFile;
    private static ArrayList        mList = new ArrayList();

    public static void main( String[] inArguments )
        throws Throwable
    {
        System.out.print( "Running... " );

        int counter = 0;
        long end = System.currentTimeMillis() + TIME;
        while ( System.currentTimeMillis() < end )
        {
            load();
            ++counter;
        }

        double count = counter / ( ( double ) TIME / 1000 );
        System.out.println(
            count + "/s (" + ( 1 / count ) + ")" );
    }

    private static
    void load()
        throws Throwable
    {
        mFile = new RandomAccessFile( "/proc/meminfo", "r" );

        parse();

        mFile.close();
    }

    private static
    void parse()
        throws Throwable
    {
        // Skip the old header.

        mFile.readLine();
        mFile.readLine();
        mFile.readLine();

        // Store the values.

        mList.clear();

        store(); // total
        store(); // free
```

```
        store(); // shared
        store(); // buffers
        store(); // cached
        store(); // swap cached
        store(); // active
        store(); // inactive
        store(); // high
        store(); // high free
        store(); // low
        store(); // low free
        store(); // swap
        store(); // swap free
    }

    private static
    void store()
        throws Throwable
    {
        // Get the next value.

        String line = mFile.readLine();

        // Skip fixed name length.

        int i = 14;

        // Skip column padding.

        while ( line.charAt( i ) == ' ' )
        {
            ++i;
        }

        // Find the end of the value.

        int j = line.indexOf( ' ', i );

        // Store the value as a String.

        mList.add( line.substring( i, j ) );
    }
}
```

We expect that this implementation will perform reasonably well. However after the first run we see that the code only renders 85 loads per second at 100% CPU utilization. The problem with this implementation is that it does not take into account what is going on inside the kernel's /proc file system or Java's RandomAccessFile readLine method.

As stated earlier, each time a read operation is performed on the /proc file system, the kernel must produce the entire set of data provided in the file. This means that if a request is made for the entire file or only a single byte, the kernel must gather the entire set of values for the file. Unfortunately the readLine method in the RandomAccessFile class gathers the characters of a line one character at a time. This means that the /proc file is generated as many times as there are bytes in the file. This is obviously extremely inefficient no matter what programming language is used.

To get around this problem, we simply need to read the file once in whole and then parse its contents. We do this by adding a byte buffer and changing the parsing code as shown in Figure 2. The result of this change is a gathering rate of 4,173 times per second, or a 4,809% increase in performance.

Figure 2. Using block reads.

```
import java.io.*;
import java.util.*;

public class memoryfilebuffer
{
    private static final int TIME = 5000;
```

```
private static RandomAccessFile mFile;
private static ArrayList        mList = new ArrayList();
private static byte[]           mBuffer = new byte[ 4096 ];
private static int              mOffset;

public static void main( String[] inArguments )
    throws Throwable
{
    System.out.print( "Running... " );

    int counter = 0;
    long end = System.currentTimeMillis() + TIME;
    while ( System.currentTimeMillis() < end )
    {
        load();
        ++counter;
    }

    double count = counter / ( ( double ) TIME / 1000 );
    System.out.println(
        count + "/s (" + ( 1 / count ) + ")" );
}

private static
void load()
    throws Throwable
{
    mFile = new RandomAccessFile( "/proc/meminfo", "r" );
    mFile.read( mBuffer );

    parse();

    mFile.close();
}

private static
void parse()
{
    mOffset = 0;

    // Skip the old header.

    while ( mBuffer[ mOffset++ ] != '\n' );
    while ( mBuffer[ mOffset++ ] != '\n' );
    while ( mBuffer[ mOffset++ ] != '\n' );

    // Position to the first value.

    mOffset += 14; // 'MemTotal:     '

    // Store the values.

    mList.clear();

    store(); // total
    store(); // free
    store(); // shared
    store(); // buffers
    store(); // cached
    store(); // swap cached
    store(); // active
    store(); // inactive
    store(); // high
    store(); // high free
    store(); // low
    store(); // low free
    store(); // swap
    store(); // swap free
}

private static
void store()
{
    // Skip column padding.

    while ( mBuffer[ mOffset ] == ' ' )
    {
        ++mOffset;
    }

    // Find the end of the value.
```

```
        int offset = mOffset:

        while ( mBuffer[ mOffset ] != ' ' )
        {
            ++mOffset;
        }

        // Store the value as a String.

        mList.add(
            new String( mBuffer, offset, mOffset - offset ) );

        // Skip to the next value.

        mOffset += 18; // ' KB\nXXXXXXXXX:    '
    }
}
```

Now we will examine the approach used to extract the values from the buffer. We are using standard methods for locating characters and creating strings. However what isn't obvious is that we are incurring overhead for implicit Unicode character encoding when we extract bytes from the buffer to create a string. Given that we already have the character data in the buffer, and given that the /proc data is always stored in the standard ASCII character range 0-127, there is no reason for us to convert the data from the buffer into Unicode strings. Instead we need to store the offset and length of the data in the buffer. We can do this with a couple of integer arrays. These changes to our parsing code eliminate the memory allocation and character encoding that is occurring now, and will eliminate the need for the character-to-byte translation that would have been necessary prior to transmission. The simple modifications to our parsing code displayed in Figure 3 yields another 236% increase in performance, or a monitoring rate of 14,031 times per second.

Figure 3. Removing unwanted memory allocation and character encoding.

```
    private static int[]    mOffsets = new int[ FIELDS ];
    private static int[]    mLengths = new int[ FIELDS ];
    private static int      mField;

    void parse()
    {
        mOffset = 0;
        mField = 0;

        // Skip the old header.

        while ( mBuffer[ mOffset++ ] != '\n' );
        while ( mBuffer[ mOffset++ ] != '\n' );
        while ( mBuffer[ mOffset++ ] != '\n' );

        // Position to the first value.

        mOffset += 14; // 'MemTotal:     '

        // Store the values.

        store(); // total
        store(); // free
        store(); // shared
        store(); // buffers
        store(); // cached
        store(); // swap cached
        store(); // active
        store(); // inactive
        store(); // high
        store(); // high free
        store(); // low
        store(); // low free
        store(); // swap
```

```
        store(); // swap free
    }

    private static
    void store()
    {
        // Skip column padding.

        while ( mBuffer[ mOffset ] == ' ' )
        {
            ++mOffset;
        }

        // Find the end of the value.

        int offset = mOffset;

        while ( mBuffer[ mOffset ] != ' ' )
        {
            ++mOffset;
        }

        // Store the value's offset and length.

        mOffsets[ mField ] = offset;
        mLengths[ mField++ ] = mOffset - offset;

        // Skip to the next value.

        mOffset += 18; // ' KB\nXXXXXXXXX:    '
    }
```

Given that the /proc file system files are mapped to memory rather than disk, and because the code for opening and closing files is usually highly optimized on any operating system, one might assume that the overhead associated with opening and closing the file would not contribute a significant amount of time to the gathering process. However this is not true.

Figure 4. Leaving the file open.

```
    public static void main( String[] inArguments )
        throws Throwable
    {
        System.out.print( "Running... " );

        mFile = new RandomAccessFile( "/proc/meminfo", "r" );

        int counter = 0;
        long end = System.currentTimeMillis() + TIME;
        while ( System.currentTimeMillis() < end )
        {
            load();
            ++counter;
        }

        mFile.close();

        double count = counter / ( ( double ) TIME / 1000 );
        System.out.println( count +
            "/s (" + ( 1 / count ) + ")" );
    }

    private static void load() throws Throwable
    {
        mFile.read( mBuffer );
        mFile.seek( 0 );

        parse();
    }
```

Our final modification is to keep the file open across multiple requests. To do this we simply move existing code from the load method to the main method, and add a seek call as shown in Figure 4. The result of this modification yields an

additional 141% increase in performance. Combined with the other three increases, the gathering rate is now 33,855 times per second or a 39,729% increase in performance over the first implementation. This translates to approximately 2.95 one hundred thousandths (0.000295) of the CPU per request. In other words approximately 5 seconds of CPU time per hour at a monitoring rate of 50 times per second. We have verified this result on the previously described machine.

6 Results

Tables 2 and 3 present the results of applying the described techniques to the five specified /proc files using the Java and C programming languages. The *Loads/s* and *Parses/s* columns represent the number of times the file can be loaded or parsed per second using 100% of the CPU. The parsing times are dependent on the complexity and size of the files. The *Total/s* column represents the combined loading and parsing rate for each file. The performance numbers in these tables do not take into account the time needed by the consolidation and transmission stages, where equal attention should be paid to performance and efficiency.

Table 2. Gathering rates for Java.

Monitor	Loads/s	Parses/s	Total/s
/proc/stat	31,297	240,871	27,954
/proc/loadavg	136,627	1,066,010	132,384
/proc/meminfo	39,727	262,371	35,265
/proc/net/dev	49,921	162,026	39,038
/proc/uptime	165,885	1,202,597	161,634

Table 3. Gathering rates for C.

Monitor	Loads/s	Parses/s	Total/s
/proc/stat	34,365	289,318	31,255
/proc/loadavg	205,795	1,287,964	188,623
/proc/meminfo	44,952	319,070	39,980
/proc/net/dev	58,108	207,342	46,291
/proc/uptime	203,912	1,746,182	197,704

It should be noted that in both Java and C implementations, the parsing rates are substantially higher than their corresponding loading rates. At the same time, the overall rate differences between the Java and C implementations are relatively small. This indicates that the monitoring throughput is now more dependent on the efficiency of the kernel code for the /proc file system, than it is on the performance differences of the implementation languages. Assuming a monitoring rate of 1 time per second, Table 4 lists the projected CPU savings that would be achieved by using C rather than Java. The values are computed by determining the time per call for each monitor and then taking their differences. The difference amounts to a combined savings of about 1 second per day.

Table 4. Performance comparison.

Monitor	Per Call	Hourly	Daily
/proc/stat	3.778e-6 seconds	0.014 seconds	0.326 seconds
/proc/loadavg	2.252e-6 seconds	0.008 seconds	0.195 seconds
/proc/meminfo	3.344e-6 seconds	0.012 seconds	0.289 seconds
/proc/net/dev	4.014e-6 seconds	0.014 seconds	0.347 seconds
/proc/uptime	1.129e-6 seconds	0.004 seconds	0.098 seconds

7 Conclusion

Optimizing takes times and experimentation, and isolating poorly performing code always involves multiple implementations. The best way to optimize any code is to identify and replace the worst performing code first. By resolving the most inefficient pieces first, we are more able to clearly identify the medium and smaller performance barriers that remain.

By taking this approach, this paper has demonstrated that the Java language can be used efficiently for high performance monitoring on the nodes of Linux clusters. In the process it has presented the following techniques for an efficient implementation:

- using the /proc file system,
- reading /proc files as a block rather than by lines or characters,
- keeping /proc files open between reads.
- eliminating unnecessary data conversions,
- consolidating data on the node,
- transmitting data in compressed form,
- being aware of language or library related performance issues,
- being aware of inefficiencies in specific kernel versions.

It has been shown that kernel modules or patches are not a requirement for high performance monitoring. This is important because it provides a greater degree of portability between Linux versions and distributions, and a greater choice of monitor implementation languages. The performance of the /proc file system however is very dependent on the efficiency of the kernel code that produces the file, and a proper understanding of the mechanisms involved can greatly affect the performance of monitors written in any language.

References

[1] Linux NetworkX, ClusterWorX: Cluster Management Solution. Available at: http://www.linuxnetworx.com/ news/pdf/whitepaper_cwx.pdf

[2] Ron Minnich and Karen Reid, Supermon: High performance monitoring for Linux clusters. *The Fifth Annual Linux Showcase and Conference, Oakland, CA* Nov, 2001.

[3] lm_sensors project, Available at http://www2.lm-sensors.nu/~lm78/.

[4] T. J. Killian. Processes as Files. *Proceedings of the USENIX Software Tools Users Group Summer Conference, pp 203-207*, June 1984.
[5] R. Faulkner and R. Gomes, The Process File System and Process Model in UNIX System V. *USENIX Conference Proceedings*, January 1991.
[6] Rajkummar Buyya, PARMON: a portable and scalable monitoring system for clusters. *Software--Practice and Experience, John Wiley and Sons, Ltd*, 2000.
[7] Eric Anderson and Dave Patterson, Exensible, Scalable Monitoring for Clusters of Computers. *The proceeeding of the 11th Systems Administration Conference (Lisa '97)*, Oct. 1997.
[8] T. Roney, A. Bailey, and J. Fullop, *Cluster Monitoring at NCSA. Linux Revolution Conference*, June 2001. Available at: http://www.ncsa.uiuc.edu/Divisions/CC/systems/LCI-Cluster-Monitor.html
[9] Z. Liang, Y. Sun, C. Wang, ClusterProbe: An Open, Flexible and Scalable Cluster Monitoring Tool. *IEEE Int. Workshop on Cluster Computing, pp. 261-268*, 1999.
[10] Quadrics RMS Pandora Reference Manual, Available at: http://www.quadrics.com.