

# Evaluating compression as an enabler for centralised monitoring in a Next Generation Network

Fred Otten, Barry Irwin and Hannah Slay  
Security and Networks Research Group (SNRG)  
Department of Computer Science  
Rhodes University, Grahamstown, South Africa  
Tel: 046 6038291 Fax: 046 6361915  
g05o5894@campus.ru.ac.za, b.irwin@ru.ac.za, h.slay@ru.ac.za

**Abstract**—With the emergence of Next Generation Networks and a large number of next generation services, the volume and diversity of information is on the rise. These networks are often large, distributed and consist of heterogeneous devices. In order to provide effective centralised monitoring and control we need to be able to assemble the relevant data at a central point. This becomes difficult because of the large quantity of data. We also would like to achieve this using the least amount of bandwidth, and minimise the latency. This paper investigates using compression to enable centralised monitoring and control. It presents the results of experiments showing that compression is an effective method of data reduction, resulting in up to 93.3 percent reduction in bandwidth usage for point-to-point transmission. This paper also describes an architecture that incorporates compression and provides centralised monitoring and control.

**Index Terms**—Network Monitoring, Data compression, Next generation networks

## I. INTRODUCTION

Next Generation Networks (NGN) are large, integrated networks offering many different services for their users. They consist of different platforms such as smartphones, Voice over IP, wireless networks, and a variety of other applications such as web servers, databases, mail servers and proxies. They offer services such as video calls, voice calls, internet access and email.

Such networks produce large amounts of diverse information, such as system logs, web server logs, traffic captures and call logs, which needs to be monitored for different purposes. These purposes include: security purposes such as detecting worms, viruses and attacks on the network; ensuring proper use and compliance to set policies [11]; satisfying regulations which require accountability, monitoring and storage of records; getting information about the users and use of a service for competitive gain [13]; and attaining usage information so that the users may be billed appropriately. The information needs to be gathered and collated for reports and further analysis. Related information may come from both legacy and newer technologies. This presents a problem for

This work was undertaken in the Distributed Multimedia Centre of Excellence at Rhodes University, with financial support from Telkom SA, Business Connexion, Comverse, Verso Technologies, Tellabs, StorTech, THRIP and the National Research Foundation.

data collation. The large quantity of information, difficulty in collation, explosive growth of the volume, diversity of information involved and the necessity of analysis motivates the proposal of a centralised monitoring approach [9].

Centralised monitoring and control offers an attractive solution, reducing redundancy, providing a comprehensive view of the network and maximising the effectiveness of the security and system administration teams by enabling the automation of some of the more tedious correlation and reporting processes. Centralised monitoring provides administrators with a holistic view of the network and its user, which in turn supports effective monitoring and control [9], [12].

The primary problem with the centralised approach is the large volume of data which needs to be sent to the central point. Samples or summaries may be sent, however this will not comply with some regulations [13] and is not guaranteed to provide an accurate view of the network state. Data compression is designed to reduce the size of a given payload. This means that we could use data compression to deal with the large volume of data associated with the monitoring of a NGN if the latency due to compression and decompression and the resources used are acceptable.

This paper examines the use of compression as a means of data reduction. It begins by surveying different types of compression algorithms and then presents the results of an experiment conducted to evaluate the effectiveness of compression, applying it to different scenarios. Finally, it presents a new architecture for monitoring and control which integrates compression, concluding with a summary of results.

## II. RELATED WORK

Generally there are four different types of redundancy which may be found in any given data source without explicit knowledge of how the data is interpreted. These are: distribution of characters; repetition of characters; high usage of certain patterns or sequences of characters; and positional redundancy of characters [5]. Data compression exploits this redundancy to compress the data source and obtain a payload whose size is as small as possible. There are many different compression programs available. Some compressors are block based, ie. they divide the data into

blocks and then perform compression on each block; some are dictionary based ie. they replace words with references to a dictionary; and some use statistical models to perform analysis and predict future characters, reconstructing the original data. Most data compression programs use a combination of these techniques to achieve better compression. Data compression is often improved using transformations (such as the Burrows-Wheeler transform [2]) and normalizations (such as the BJC filters in *7zip* [17]).

Lempel-Ziv based algorithms are very popular. LZ77 [19] and LZ78 [20] are two such compression algorithms. They are based on papers written by Lempel and Ziv in 1977 and 1978 respectively. DEFLATE [3] (which is used in *zip* and *gzip*) is a common LZ77 based algorithm. It uses a combination of the LZ77 algorithm and Huffman coding [6].

Techniques such as Huffman coding, arithmetic encoding [18] or range encoding [7] may be used to construct statistical codes using the distribution of characters. Smaller codes are constructed for more frequent occurring characters, improving data compression.

The compression programs we will investigate in this paper are *ppmd*, *7zip*, *zip*, *gzip*, *bzip2*, *lzop* and *arj*. *ppmd* uses the "Prediction by Partial Matching with Information Inheritance" (PPM-II) algorithm. In this algorithm, a statistical model is constructed and used to predict the next character [15]. *7zip* uses a Lempel Ziv based algorithm with Markov modeling [10]. *zip* and *gzip* [4] use the DEFLATE algorithm. *zip* is able to use a large array of algorithms, however the default, which is available on all implementations, is the DEFLATE algorithm. *bzip2* uses a combination of a Burrows-Wheeler transform, a Move-to-front transform and Huffman encoding. This is known as a block sorting algorithm. *lzop* is a fast compression program which uses the LZO family of algorithms. The LZO algorithms are a family of block based algorithms based on the LZ78 algorithm [8]. *arj* is an old compression program which uses an LZ77 based algorithm with hashing [16]. This sample of programs are available on many different platforms and employ a vast range of different compression techniques.

### III. EXPERIMENT

#### A. Overview

Most of the data used for monitoring is stored in the form of text log files. This data needs to be sent to a central point for centralised monitoring. Data compression techniques may be used to remove the redundancy in data and decrease the size of the payload which needs to be sent. Compression often results in an 80 to 90 percent reduction in file size for text files. This makes it an attractive option for data reduction given that the latency due to compression and decompression and the resources used are not too high. Benchmarking figures are available for the Calgary Corpus and Hector corpus [1]. These results show the compression ratio achieved, but do not tie it to the time taken (latency) and the resources used. This experiment aims to determine whether compression will provide an effective means of reducing the quantity of data with an acceptable latency. It also aims to show which algorithms would be best suited for the different situations in centralised monitoring.

Position	Program	Compression level	Time
1	lzop	6	0.08
7	arj	1	0.09
8	gzip	2	0.17
11	zip	2	0.18
29	ppmd	2	0.96
35	7zip	1, 0	1.21
48	bzip2	1	2.72

TABLE I  
FASTEST COMPRESSION TIMES

Position	Program	Compression level	Time
1	lzop	9	0.05
9	arj	1	0.07
11	gzip	9	0.1
12	zip	9	0.1
32	7zip	7, 1	0.31
44	bzip2	1	0.57
53	ppmd	2	1.07

TABLE II  
FASTEST DECOMPRESSION TIMES

#### B. Method

We take a number of log files and perform compression and decompression using *ppmd*, *7zip*, *zip*, *gzip*, *bzip2*, *lzop* and *arj*. The files are compressed and decompressed five times for all compression levels available for each program. The resulting times and ratios are recorded and the means calculated. The tests are performed using Perl scripts and the timing information recorded into a file by the Unix *time* program.

#### C. Results

The tests are conducted on a number of log files: squid access logs; postfix mail logs; kernel messages; ftp logs; and generic syslogs. We also perform tests on a LibPCap packet capture file to determine how the program performs on binary packet captures. In this section we present the results for the generic syslog file since it is the most common log format and many log formats are based on it. We also comment on the effectiveness of the program on binary data.

Tables I, II, III and IV reflect the fastest compression time, decompression time, total compression time and the lowest compression ratio respectively for each compression program. All results reflected in these tables are averaged over the five iterations and are rounded to five decimal places, with times reflected in seconds (s). The compression level column shows the compression level for which the result was achieved for the specified program. The position column shows the rank achieved of that result when the results for all programs and all compression levels are ordered from fastest to slowest in the case of time and lowest to highest in the case of ratio.

Table V illustrates the difference between the average compression ratios over all compression levels of the LibPCap (binary) and syslog (text) files. This table shows a higher standard deviation for *lzop* and *ppmd* than for the other compression programs. In the case of *lzop*, there are essentially two types of compressors: the first six compression levels,

Position	Program	Compression level	Time
1	lzop	3	0.13
6	arj	1	0.16
8	gzip	2	0.27
11	zip	3	0.29
30	7zip	1, 0	1.57
33	ppmd	2	2.03
43	bzip2	1	3.29

TABLE III

FASTEST TOTAL COMPRESSION AND DECOMPRESSION TIMES

Position	Program	Compression level	Ratio
1	ppmd	10	0.05585
14	7zip	9, 2	0.07682
18	bzip2	9	0.08064
36	gzip	9	0.11531
38	zip	9	0.11533
42	arj	1	0.11760
53	lzop	8	0.14362

TABLE IV

LOWEST COMPRESSION RATIOS

which are very fast and do not achieve good compression ratios; and the last three, which are much slower and designed to achieve better compression ratios. These are grouped as *lzop(1)* and *lzop(2)* respectively in the table. These groupings show more acceptable standard deviations. In the case of *ppmd*, we notice that the results for order 2 (0.02405) are much lower than the mean for *ppmd*, so *ppmd\** excludes this outlier and yields a much lower standard deviation.

1) *7zip* and *ppmd*: As can be seen in Table IV, the newer programs such as *ppmd* and *7zip* achieve very good compression ratios. From testing, we notice that they are also resource intensive. *ppmd* can use a maximum of 128 megabytes (MB) of memory [14], at which it starts the model again. Our tests show that from around order 6 it peaks and restarts the model. *7zip* uses a binary tree or a hash chain as the data structure to find matches in the dictionary. It uses between 4.4 MB and 742 MB of memory for these data structures [17]. This is resource intensive.

Table I and Table II show that both *7zip* and *ppmd* are also slow in comparison to the other programs. *7zip* exhibits a faster decompression than compression time (as is the case with most of the programs), whereas *ppmd* is slower at decompression than compression.

We expect compression programs to perform worse on binary files than on text files. Table V illustrates the difference. It shows *ppmd* and *ppmd\** achieving the worst results. One could say that this outlier excluded in *ppmd\** is the one point where *ppmd* performs very well. This result occurs for order 2, which achieves a compression ratio of 0.21018 on the LibPCap file. This is higher than the mean ratio for all the other programs which are shown in Table V. The rest of the compression ratios achieved for *ppmd* are between 0.14359 and 0.17598. On the syslog file, order 2 yields a result of 0.18613, while the rest of the orders yield ratios between 0.05585 and 0.10166, which are nearly half of the result for order 2. We can conclude that *ppmd* does not perform well at this outlier, and that at this order *ppmd* does not perform well

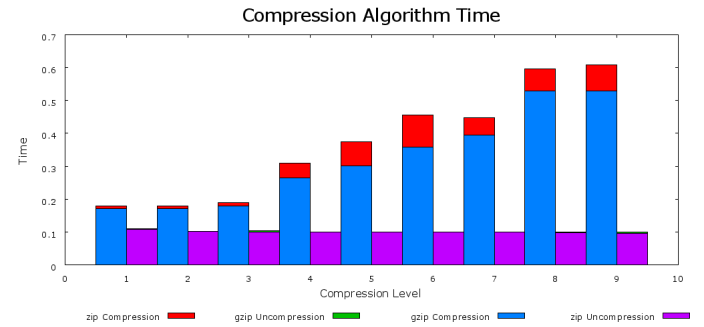
Program	Mean	Standard Deviation	Mean Ratio (LibPCap)
lzop(1)	0.00627	0.00069	0.20403
lzop	0.01871	0.01866	0.19864
7zip	0.02681	0.00599	0.11596
lzop(2)	0.04358	0.00077	0.18788
zip	0.04582	0.00894	0.17422
gzip	0.04583	0.00894	0.17422
arj	0.04756	0.00900	0.18268
bzip2	0.07258	0.00553	0.16092
ppmd	0.08232	0.01659	0.15348
ppmd*	0.08649	0.00406	0.14943

TABLE V

COMPRESSION RATIO DIFFERENCE BETWEEN LIBPCAP (BINARY) AND SYSLOG (TEXT) FILES

overall. These results show us that *ppmd* does not perform very well on the binary LibPCap files in comparison to the text syslog files.

*7zip* achieves very good results for the LibPCap file. As shown in Table V, the difference between the syslog file and the LibPCap file is not very high and *7zip* also achieves a very good compression ratio. *lzop*, which according to Table V has the smallest difference, does not achieve a good compression ratio. We can therefore say that *7zip* performs the best on LibPCap files.

Figure 1. *zip* and *gzip* compression times

2) *zip*, *gzip* and *arj*: Both *zip* and *gzip* use the DEFLATE algorithm. It is therefore not surprising to see them exhibit similar results. Figure 1 shows their respective compression and decompression times for the different compression levels. Both *gzip* and *zip* exhibit fast decompression times. We see in Table IV that *gzip* achieves a lower compression ratio than *zip*. This is because *gzip* has smaller headers than *zip*. *gzip* also achieves faster compression times than *zip*. This is clearly illustrated in Figure 1. For both programs, as the compression level increases, the decompression time stays approximately constant, while the compression time exhibits an exponential growth.

The average difference between LibPCap binary files and syslog text files overall, for all programs and compression levels, is 0.05065 and the mean of our averages in Table IV is 0.04760. *zip* and *gzip* achieve 0.04582 and 0.04583 respectively for this result, which is lower than both averages. We can thus conclude that both programs perform above average on the LibPCap binary files.

*arj*, which also uses an LZ77 based algorithm, achieves similar times to both *zip* and *gzip*. It achieves the second

fastest compression and decompression times, faster in both cases than the fastest times for *zip* and *gzip*. *arj*, however, achieves higher compression ratios than *gzip* and *zip* and exhibits average performance on the LibPCap binary file.

3) *bzip2*: *bzip2* achieves the third lowest compression ratio for a generic syslog file (this may be seen in Table IV), but is slow. Tables I, II, and III, show that *bzip2* is one of the slowest programs. It achieves the slowest compression time and the second slowest decompression time, giving it the slowest total time. *bzip2* doesn't perform as well on the LibPCap binary files as it does on the text-based syslog files. Table V shows that only *ppmd* achieves worse results for the difference between LibPCap files and syslog files than *bzip2*.

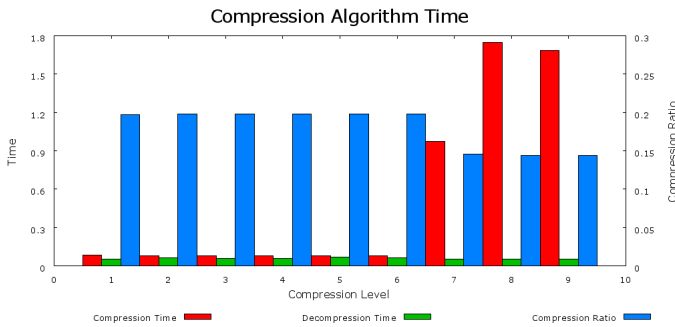


Figure 2. *lzop* compression and decompression times and resulting compression ratios

4) *lzop*: *lzop* is a very fast compression program. Tables I and II show that it achieves the fastest compression and decompression times. Figure 2 illustrates the compression times, decompression times and resulting compression ratios for the different compression levels of *lzop*. *lzop* exhibits a constant, fast decompression time and shows fast compression times for the lower compression levels (1-6). The higher compression levels (7-9) take much longer (up to 17 times longer) and result in lower compression ratios. These compression ratios, however, are still poor in comparison to the other programs such as *zip*, *gzip* and *arj* which achieve faster times for equivalent compression.

The lower compression levels of *lzop* (1-6) achieve very little difference in results between the text-based syslog and the binary LibPCap files. These are grouped as *lzop(1)* in Table V. The remaining group, compression levels 7-9, *lzop(2)*, also show good results, achieving worse results than *7zip*. The compression ratios achieved by *lzop*, however, are poor in comparison. They are between one and a half and two times the compression ratios achieved by *7zip*.

*lzop* is not a good choice unless the main consideration is speed. In this case, the first group of *lzop* algorithms - *lzop(1)* - shows good results.

#### D. Further testing

The aforementioned tests give a good picture of the performance of the compression programs. If we are going to use these programs to compress data, transmit it to another point and then decompress it, we need to investigate the latency for point-to-point transmission (ie. the sum of compression time,

decompression time and the time to transmit the compressed payload at a given rate).

In this section we present the results for tests using the first 1 MB of a syslog file. We compress and decompress it five times, measure the compression and decompression times for each and record the averages. We then calculate how long it would take to transmit the compressed payload at 10, 25 and 50 kilobytes per second (KB/s) and, hence, calculate the resulting point-to-point times.

Position	Program	Compression level	Time (s)
1	gzip	7	2.49367
6	zip	7	2.56611
9	ppmd	5	2.63836
17	arj	2	2.74389
38	lzop	7	3.39107
42	7zip	1, 0	3.39676
53	bzip2	1	4.10633

TABLE VI  
POINT-TO-POINT TIME AT 50 KB/S

Position	Program	Compression level	Time (s)
1	ppmd	6	3.93664
13	gzip	7	4.73734
17	zip	7	4.81223
25	arj	2	5.04777
30	7zip	1, 0	5.48352
35	bzip2	2	6.04590
39	lzop	7	6.21215

TABLE VII  
POINT-TO-POINT TIME AT 25 KB/S

Position	Program	Compression level	Time (s)
1	ppmd	10	6.81811
15	7zip	1, 0	11.34379
16	gzip	9	11.41094
18	bzip2	3	11.44498
22	zip	9	11.52314
33	arj	2	11.95943
51	lzop	1	14.67537

TABLE VIII  
POINT-TO-POINT TIME AT 10 KB/S

Tables VI, VII and VIII show the fastest point-to-point time for each program. The position is ranking of the time taken from the fastest (1) to the slowest (67). The compression level specifies the compression level for which the result was achieved.

Tables VI, VII and VIII show that as the transfer speed increases, the results change dramatically. It is interesting that the slower programs such as *ppmd* and *7zip* achieve good results for slower speeds. It must also be noted that the transfer times without compression are 20.48, 40.96 and 102.4 seconds at 50, 25 and 10 KB/s respectively. So thus we see between 80 and 93.3 percent decrease in the amount of time / bandwidth necessary to transfer data from point-to-point. *gzip* appears in the top three for each of the speeds. This is because it is fast and achieves a good compression ratio. *ppmd* also achieves very good results, particularly at slower speeds, but is very

memory intensive. In general we see that the programs which produce lower compression ratios (such as *7zip* and *ppmd*) work better for the slower transfer times, while the balanced programs (such as *zip* and *gzip*) produce better results at faster speeds. This makes sense, as the trade off is between the time taken to compress and decompress that data and the difference in the time taken to transfer the resulting compressed payload. *lzop* is a poor option because of the poor compression ratio achieved.

#### E. Analysis and Application

There are a number of scenarios which we need to investigate in order to decide which compression program is most applicable. They all have their own strengths and weaknesses. Some are fast, but do not achieve good compression ratios such as *lzop*, while others have high compression times, slower decompression times but achieve good compression ratios such as *7zip*. For each monitoring scenario a different compression program is likely to be more appropriate. Some of the possible scenarios include:

- Filtering logs through to the central point for analysis (latency not important, but desire to use as little bandwidth as possible)
- Real-time monitoring (minimum point-to-point time desired, using as little resources as possible)
- Quick access, storing it compressed and decompressing it for analysis (fast decompression desired, but compression time does not matter, desire to use as little bandwidth to send as possible)
- Low system time usage for compression and decompression (fast compression and decompression times, compression ratio does not matter)

In this section we take a look at these different scenarios, and choose a compression program for each.

1) *Filtering logs through to the central point for analysis:* In this case, the compression and decompression times are not very important. We desire a good compression ratio first and foremost. This would make *ppmd*, *7zip* and *bzip2* our best options, since they achieved positions one, two and three respectively in Table IV, which shows the lowest compression ratios. Since we are going to use our logs for analysis, at the central point it is desirable to have a quick decompression time. We also need to weigh up the amount of memory utilization we desire. *bzip2* is the best option if we wish to have low memory utilization, however it achieves the slowest times and highest compression ratio out of the three programs. *7zip* and *ppmd* are both quite memory intensive. *7zip* has the fastest decompression time and it achieved the best results for LibPCap data (making it rather versatile). The best option is *7zip* due to its high compression ratio, fast decompression times and versatility.

2) *Real-time monitoring:* For real time monitoring, we desire a good point-to-point time. *gzip* and *ppmd* are hence the two best options. *ppmd* is rather resource intensive, and does not achieve as good compression and decompression times as *gzip*. *ppmd* does, however, achieve a much lower compression ratio and superior point-to-point times for the slower links than *gzip*. *gzip* produces good results for the point-to-point

times (top three in all our point-to-point tests) and is not very resource intensive. This makes *gzip* our best choice for real-time monitoring.

3) *Quick access to stored compressed information:* If the compression time does not matter, but quick decompression is desired, then *7zip* is a very good option. It achieves a good compression ratio, provides quick decompression, but has high compression times. *lzop*, *gzip*, *arj*, and *zip* all have faster decompression times but achieve higher compression ratios than *7zip*. Out of these, *gzip* achieves the lowest compression ratio and *lzop* the highest (*lzop* is nearly 25 percent larger (24.548) than *gzip*). *gzip*'s compressed payload is also a third larger than *7zip*, but exhibits faster compression and decompression times. Since the compression time does not matter, our choice would be *7zip* due to the good compression ratio.

4) *Low system time usage for compression and decompression:* For quick compression and decompression without taking any regard to size, *lzop* would undoubtedly be the best choice since it exhibits both the fastest compression and decompression times. *gzip* and *arj* also perform really well in this respect and achieve lower compression ratios than *lzop*. *arj* achieves the second fastest total time out of the programs, but exhibits higher compression ratios than *gzip*. Since fast compression and decompression is desired (ie. low system time), *lzop* would be the best choice. Should the compression ratio be taken into account, *gzip* would be the best option.

#### IV. NEW ARCHITECTURE

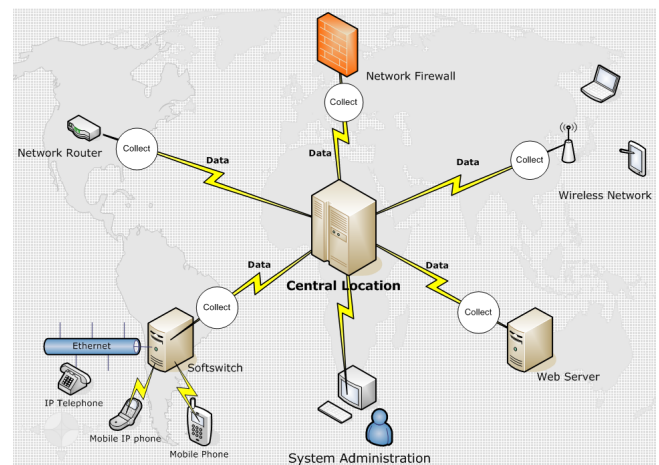


Figure 3. Architecture for centralised monitoring and control

We have evaluated compression programs and selected compression algorithms for different scenarios. The question which remains is how compression would integrate into the centralised monitoring system. We require a flexible architecture which can be molded to the particular scenario we are presented with. Otten et al. [9] presents a flexible architecture for centralised monitoring. This architecture uses collector scripts to send information to a central point via XML. The problem is that XML is verbose, and hence requires a large amount of bandwidth. We can minimise the bandwidth and latency requirements using data compression. Figure 3 illustrates our architecture with a softswitch, a wireless network, a

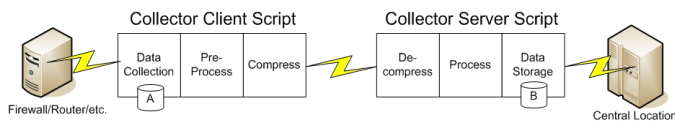


Figure 4. Integrating compression

firewall, a router and a web server. This architecture is similar to the one previously presented in [9]. We have replaced the XML representation of the data stream with a compressed data stream to minimise the bandwidth and latency requirements.

The collectors are scripts which determine what and how data is sent to the central point. Figure 4 illustrates an example of how compression may be integrated into the collector scripts. On the client side we have a router or firewall where information is located which needs to be sent to the central point (the server side). Since the collectors determine what and how the data is sent, they can be modified for use in all the different scenarios mentioned in Section III-E. Real-time monitoring, trickling data to a central point for analysis or sending sample or summaries are all possible with the collector-based architecture [9]. Certain processing or data fusion may be done before the data is sent to the central point. The collector script may even be located on a separate machine which connects to the softswitch, firewall or router and extracts the data, processing it into a single format before sending it to the central point. The decompression process may even be moved till after the data storage on the server side. This illustrates the flexibility of this architecture.

In the example given in Figure 4, data source A, on the client side, is extracted from a firewall or router. This may be a system log file. The data is preprocessed by our script. System specific information may be added or removed and then transformed into the relevant format. The transformation may use knowledge of the data format to perform reduction. The resulting data is then compressed and sent to the central point where it is decompressed. The data is then processed, data reductions expanded and interpreted into a form for analysis (in this case just restoring the original system log file). The resulting information is then stored in data source B. This may be a database or a flat-file system. The compression, decompression process may be transparent in the form of an IP-comp tunnel or an Secure Shell (ssh) tunnel with compression enabled. The scenario will dictate how the compression is embedded.

Our implementation for centralised monitoring and control uses the Twisted-Python framework for client and server scripts and ssh tunnels with *gzip* compression enabled for data transfer. This provides security as well as compression to our system.

## V. CONCLUSION

The primary problem with the centralised approach to monitoring next generation networks is the large volume of data which needs to be sent to the central point and the quantity of bandwidth available. Section III evaluates the use of a representative sample of compression programs (*ppmd*, *7zip*, *zip*, *bzip2*, *gzip*, *lzop* and *arj*), which are available on many different platforms and use a vast range of techniques for data

compression, to reduce the transmitted payload. It shows that data compression is an effective method of data reduction, resulting in between 78.8 and 94.4 percent reduction in data size and between 80 and 93.3 percent reduction in point-to-point times. Programs such as *gzip* use very little system time and resources and provide good compression ratios. *7zip* and *gzip* emerge as the best compression programs in the given monitoring scenarios. We also find that the latency due to compression and decompression is negligible. Section IV shows that data compression can be easily integrated into a flexible, cross-platform, centralised monitoring and control architecture. We can thus conclude that compression may be used to effectively aid centralised monitoring and control in next generation networks and services by dealing with the large volume of information.

## REFERENCES

- [1] T. Bell, I. H. Witten, and J. G. Clearly. Modeling for text compression. *ACM Computing Surveys*, 21(4):557–589, December 1989.
- [2] M. Burrows and D. J. Wheeler. A Block-sorting Lossless Data Compression Algorithm. Research report, Digital Systems Research Center, May 1994.
- [3] P. Deutsch. DEFLATE Compressed Data Format Specification version 1.3. Request for Comments: 1951, May 1996.
- [4] P. Deutsch. Gzip file specification version 4.3. Request for Comments: 1952, May 1996.
- [5] P. G. Howard. *The Design and Analysis of Efficient Lossless Data Compression Systems*. PhD thesis, Department of Computer Science, Brown University, June 1993.
- [6] D. A. Huffman. A Method for Construction of Minimum-Redundancy Codes. In *In Proceedings of the IRE*, volume 40, pages 1098–1101, September 1952.
- [7] G. N. N. Martin. Range encoding: an algorithm for removing redundancy from a digitised message. In *Video and Data Recording Conference*, July 1979.
- [8] M. F. X. J. Oberhumer. LZ0 - A real-time data compression algorithm. <http://www.oberhumer.com/opensource/lzo>, October 2005.
- [9] F. Otten, B. Irwin, and H. Slay. The need for centralised, cross platform information aggregation. In *Proceedings of ISSA 2006*, July 2006.
- [10] I. Pavlov. 7z format. <http://www.7-zip.org/7z.html>, 2006.
- [11] S. Read-Miller. Security Management: A New Model to Align Security With Business Needs. Computer Associates White Paper, 2005.
- [12] S. Read-Miller and R. A. Rosenthal. Best Practices for building a Security Operations Center. Computer Associates White Paper, 2005.
- [13] A. Sah. A New Architecture for Managing Enterprise Log Data. In *LISA '02: Proceedings of the 16th USENIX conference on System administration*, pages 121–132, Berkeley, CA, USA, 2002. USENIX Association.
- [14] D. Shkarin. *PPMd 9.1-12 System Manual Page*.
- [15] D. Shkarin. PPM: One step to practicality. In *Proceedings of 12th IEEE Data Compression Conference (DCC)*, pages 202–211, 2002.
- [16] Unknown. ARJ Technical Information. <http://datacompression.info/ArchiveFormats/arj.txt>, April 1993.
- [17] Unknown. 7zip system documentation. DOCS/MANUAL/switches/method.htm, November 2005.
- [18] I. H. Witten, R. M. Neal, and J. G. Clearly. Arithmetic Coding for Data Compression. *Communications of the ACM*, 30(30):520–540, June 1987.
- [19] J. Ziv and A. Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions of Information Theory*, 23(3):337–343, May 1977.
- [20] J. Ziv and A. Lempel. Compression of Individual Sequences via Variable-Rate Coding. *IEEE Transactions of Information Theory*, 24(5):530–536, September 1978.

**Fred Otten** is a student at Rhodes University in the Department of Computer Science. He is currently reading for his Masters in Science and has a keen interest in real-time multimedia, network security, real-time monitoring, computer graphics and security visualisations.