Filtering Useless Data at the Source

Pablo Pessolani[1], Constanza Quaglia[1], and Ramón Nou[2]

[1]Departamento de Ingeniería en Sistemas de Información
Universidad Tecnológica Nacional - Facultad Regional Santa Fe
Santa Fe, Argentina
{ppessolani,cquaglia}@frsf.utn.edu.ar

[2]Barcelona Supercomputing Center
Departamento de Arquitectura de Computadores
Universitat Politécnica de Catalunya. Barcelona, España
ramon.nou@bsc.es

Abstract. There are some processing environments where an application reads remote sequential files with a large number of records only to use some of them. Examples of those environments are servers, proxies, firewall and intrusion detection log analysis tools, sensor log analysis, large scientific datasets processing, etc.
To be processed, all file records must be transferred through the network, and all of them must be processed by the application. Some of the transferred records would be discarded immediately by the application because it has no interest in them, but they just consumed network bandwidth and operating system's cache buffers.
This article proposes to filter records from the source of data but without changing the application. Those records of interest will be transferred without modifications but only references to the other records will be transferred from the source to the consuming application. At the application side, the sequence of records is rebuilt, keeping the content of records of interest and filling the others with dummy values which will be discarded by the application. As the number and length of records are preserved (and therefore the file size too), it is not necessary to modify the application. Once a filtering rule is applied to a file, only the useful records and references to unuseful ones will be transferred to the application side reducing network usage, transfer time, and cache utilization. A modified (but compatible) version of NFS protocol was developed as a proof of concept.

Keywords: Logging, Network File System, NFS

1 Introduction

Some remote applications, servers, network devices, and IoT sensors produce a large number of data which will be processed by other applications in a central location. Online transactional applications, backup servers, mail servers, web

server, network proxies, routers, firewalls, IDS/IPS, environment sensors, etc. are a sample of that. Often, an application is used to analyze the content of these data to search records of interest as suspicious transactions, abnormal behavior, patterns of security attacks, malformed network packets, excessive temperature detected by a sensor, etc. To perform the data analysis the complete dataset must be transferred from the source to the central application. As the application would not be interested in the content of all data records, several records will be transferred from the source to the client but later, they would be ignored/discarded by the application. This behavior misspends network bandwidth, operating system cache buffers and, in some cases, increases the transfer time. An additional drawback occurs in cases where the network is leased and and the services are paid for each packet or amount of transferred data.

This article proposes to filter records from the source of data but without changing the application. Those records of interest will be transferred without modifications but only references to the other records will be transferred from the source to the application. At the application side, the sequence of records is rebuilt, keeping the content of matching ones and filling the others with dummy values which will be discarded by the application. As the number and length of records and the file size are preserved, it is not necessary to modify the application.

The *"filtering useless data at the source"* approach is not new. It is often used by Client/Server applications where the Client request data using and Database (like SQL) query to filter those records of interest. Then, the Database Server replies sending only the requested records to the Client application.

As a proof of concept of the *"filtering useless data at the source"* approach, a modified (but compatible) version of NFS protocol version 3 was developed, based on UNFS3 [1] and HSFS [2]. The same approach could be used on other network or distributed file systems, FTP-like servers or storage devices. Seeing the processing power that controllers of storage devices currently have, a portion of these power could be used to perform filtering at the source [3].

The remainder of this paper is organized as follows. Section 2 describes the authors' motivation for this project. Section 3 outlines FNFS design and implementation. Section 4 presents some experimental results. Finally, section 5 presents conclusions and future work.

2 Motivation

Data traffic reduction is essential when it comes to high latency links, limited bandwidth, high packet loss or when the system uses a paid communication service in which amount of transferred data impacts on the costs, or when an autonomous device needs to save energy to keep its batteries charged for as much time. Low Earth Orbit (LEO) satellite and mobile networks like 4G are examples of leased links with appreciable latency, eventually packet loss, and limited bandwidth. IoT devices must be specially considered, as they are being increasingly used for different purposes. These devices normally have a low cache

or buffer storage (due to having low memory in general) and energy usage is a critical issue since they are battery-powered. In this sense, reducing packet transmission also helps to reduce battery usage.

This article proposes a methodology to mitigate these negative issues of data transfer through these kinds of networks.

3 Filtered Network File System

This section presents FNFS design goals and implementation details that includes:

1. How to specify filtering rules outside the application which consumes the data.
2. How the data are encoded and transferred from the Server to the Client.
3. How data compression can be made on the transferred data.

The prototype implements Filtered NFS (FNFS) using a user-space NFS server [1] and client [2] as base source code. Even though the prototype was implemented using user-space NFS server and client, the same approach would be applied to kernel versions and to other network/cluster filesystems.

Although FNFS can be used in production environments, it was developed as a proof of concept of the "filtering useless data at the source" approach. Currently, it has some constraints as it can only filter sequential ASCII text files.

3.1 FNFS Design goals

The following are the properties of FNFS established as design goals:

1. *Compatibility:* Keep client and server compatibility with unmodified NFS servers and clients.
2. *UID and File specification:* The filters will be applied considering the filename (complete path) and the UID of the requester client.
3. *External filtering rule specification:* The application does not need to be modified. Therefore, an external filtering rule specification mechanism must be used.
4. *Efficiency:* The filtering mechanism must have high efficiency related to network bandwidth usage, count of packet transfers, cache buffers savings and total transfer time.
5. *Simple filtering rule specification:* Filtering rules must be specified by a simple and well-known language.
6. *Fixed and Variable length records:* The length of records must be kept and it must support fixed length records and variable length records (Line Feed terminated).
7. *Discretional data encoding:* Encoding only will be applied if appropriate.

8. *Include compression:* To improve network bandwidth and packet transfer savings, apply data compression on records of filtered files.

FNFS lets specify filtering rules outside an application that will be applied by the Server before sending file records to the Client. This feature is very useful because different filtering rules could be applied for each user/application which uses the data or when modifying the application is expensive or when the application source programs are not available.

As it is well known, NFS is a stateless file-server protocol. The Client side of FNFS gets the filtering rules, encodes them on each read request afterwards it sends them to the Server. The Server decodes the *Read Request* with the filtering rules and applies them to the file records. Those records which match the filtering rules are sent from the Client to the Server without modifications. For those records which do not match the filtering rules, only the length and position of them are encoded and sent to the Client.

Later, when the Client receives the reply, it decodes the data received from the server, and it rebuilds the sequence of file records filling the not-matching records with specified dummy data that will be ignored or discarded by the application. An example can clarify FNFS operation. Suppose that the application only processes records with the text **"ARGENTINA"** in position 10, discarding the other records.

```
char *country; // point to the country name first character
country = &record[10]; // record just read
if (strncmp(country, "ARGENTINA", 9) == 0) {
    // process the matching record
} else {
    // discard the record
}
```

When FNFS Client receives the code of a not-matching record from the Server, it creates a dummy record with dummy text in position 10 as **"123456789"** whose will be discarded by the application *if* sentence.

As the application processes the same number of records as the original file and record lengths are preserved, there is no need to modify it. In addition to FNFS data filtering features, data compression was added to improve the usage of network bandwidth and to reduce the transfer time. Data compression is a feature that the user can disable. In some use cases, it is not desired due to the higher CPU and memory usage required (i.e. IoT devices).

Data encryption is another feature considered to be added in the near future. The current version presented here only supports sequential ASCII text files with fixed or variable record length and text filters.

3.2 Filtering Specifications

As was mentioned earlier, filters are specified outside the application. FNFS adopts a simple approach by using shell environment variables.

- *FNFS_FILTER:* Specifies if the filter will be applied (YES/NO).
- *FNFS_FILENAME*: The pathname of the file to be read and filtered.
- *FNFS_RLEN*: The record length. It must be a positive number for fixed length records or zero for variable length records.
- *FNFS_FILTER*: The text search on each record.
- *FNFS_MATCH*: If the value is "YES" it means that the matching registers will be transferred to the Client. If the value is "NO", unmatching registers will be transferred to the client.
- *FNFS_DUMMY*: It specifies the character which replaces all characters within useless registers. i.e. "#".
- *FNFS_COMPRESS*: Specifies if compression will be applied (YES/NO).

These environment variables must be set and exported before starting FNFS Client. Once the Client checks correct values, it codifies them into newly added fields of the NFS read request.

3.3 Data Encoding

Data encoding must meet design goals #4, #6, #7. The server sends to the client a block of data which consists of a variable size list followed by a sequence of matching records as shown in Figure 1.

Figure 1 presents the data encoding of a block of 10 fixed length records of 80 bytes each (records #0 to #9). The records that match the filtering rules are #2, #4, #5, #6, #8 and #9. The first two rows represent a list of 10 items, one for each record. Each list element with a value greater than zero represents the record length of a non-matching record. If the value is 0, it means that the record in this position matches the filtering rule, and it is stored in order and without modifications within the block. Figure 1 represents 10 records of 80 bytes or 800 bytes, but only 6 records of 80 bytes (480 bytes) plus the size of the array (10 elements of 4 bytes) are sent to the client, saving about 35% of data to be sent through the network. Figure 2 presents the data block encoding of variable length records terminated with Line Feed character.

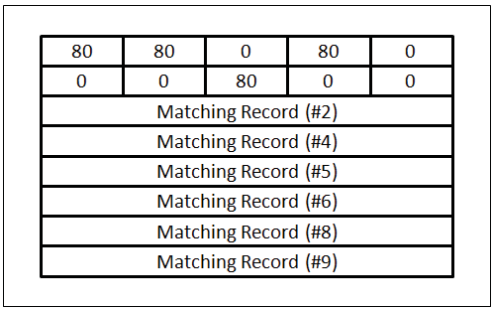


Fig. 1: Fixed Length Records Data Block Encoding

The last character LF on each matching record is a Line Feed ASCII code. It is used by the FNFS client to compute the record length. Figure 2 represents the encoding of a block of 10 records totalizing 521 bytes, but only 6 records totalizing 348 bytes plus the size of the array (10 elements of 4 bytes) are sent to the client, saving about 26% of data to be sent through the network.

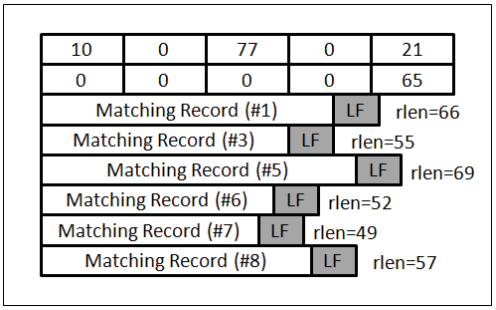


Fig. 2: Variable Length Records Data Block Encoding

Let *rlen* the length (bytes) of each record and *nrecs* the total amount of records of the file.

For fixed length records, without filtering, the total bytes is defined as:

$$T_{bytes} = rlen * nrecs \tag{1}$$

With filters, and a matching rate k of the filtering rule ($0 \leq k \leq 1$), the number of bytes to be transferred is:

$$F_{bytes} = (rlen * nrecs * k) + (nrecs * 4) = n_recs * ((rlen * k) + 4) \tag{2}$$

For variable length records, without filtering:

$$T_{bytes} = \sum rlen_i \tag{3}$$

With filters and $K_i = \{0, 1\}$ where 1 means that the record match:

$$F_{bytes} = \sum (rlen_i * K_i) + (nrecs * 4) \tag{4}$$

Only if $F_{bytes} < T_{bytes}$ the block is encoded by FNFS. The saving ratio is $1 - \frac{F_{bytes}}{T_{bytes}}$.

3.4 Data Compression

Data filtering reduces the total data sent from Server to Client. Applying compression to encoded data introduces another level of efficiency improvement on network usage. The simple additions of compression functions to the filtering code produce important savings.

The resulting block of encoded data on the Server is compressed using the zlib[4] compression library functions.

4 Evaluation

Several benchmarks have been performed with simulated networks, considering different latency and bandwidth scenarios, and with and without data compression. The tests consist of filtering records without the occurrence of an ASCII string. Only records with the occurrence of the string will be transferred without modification. On the server-side, several files were created with different rate of occurrence of the string to be filtered. The x-axis in charts represents the hit rate of the filtering rule. A hit rate of 128 means that the probability of string matching is 1/128, and so on. A hit rate of 1 means that the probability of string matching is 1/1 (the entire file).

For each simulation, the charts show the amount of transmitted packages and transmission time, under the specified conditions. Each line is labeled as follows:

- *NFS*: NFS protocol without modifications.
- *FNFS*: Filtered NFS without compression.
- *FNFS w/c*: Filtered NFS with compression.
- *GZIP*: NFS standard protocol with the file previously compressed with GZIP at the server and decompressed at the client.

Simulations were made using the Traffic Control tool (*tc*) on the server side. *tc* allows modifying traffic parameters like latency, bandwidth limit, and error rate. In this way, the server was set to simulate VSAT, Low-Earth Orbit (LEO) and Mid-Earth Orbit (MEO) satellite, and mobile 4G communication behaviours.

4.1 VSAT

VSATs (Very Small Aperture Terminals) consist of small satellite stations, that work with Geostationary Satellites (GEO)[5]. Each VSAT station can be configured in point-to-point or mesh topology. Usually, they are configured in a star topology with a special station with a high-gain antenna, called Hub. In this configuration, the signal must go through two round trips to the satellite increasing the latency time.

4.2 LEO and MEO satellite constellations

A constellation of LEO satellites is placed near the surface of the earth (about 2000 to 5000 Km). Due to their closeness, stations do not need much power, and the communication delay is around a few milliseconds (30-50 [ms])[6]. MEO satellite constellation moves across the sky at a height of 8000 Km with a delay of 70 to 150 [ms].

4.3 Mobile 4G communications

4G mobile networks[7] are intended to provide services like video and voice streaming and mobility support. Packet loss must be especially considered here, due to the kind of services these networks are thought to bring.

4.4 Simulation Results

To evaluate the performance and the operation of FNFS, two nodes were used with the following hardware configurations: Server) AMD A6-3670, 2.7 GHz, 8GB RAM. Client) Intel(R) Celeron(R) CPU G1820, 2.7 GHz, 4 GB RAM. The nodes were connected through a dedicated 1 Gbps LAN Switch. Both nodes run Debian 9.4 (called "stretch") with a Linux kernel version 4.9.88.

The benchmarks performed to evaluate the amount of packet transferred which would affect the network and battery usage, eventually the cost of the transfer, and the total transfer time. Different hit rates of the filtered string, between 0 (no matches) and 1 (100% matches) were used for the benchmarks. Table 1 shows the configuration parameters (delay, bandwidth and packet loss) considered for each technology. Fig. 3 presents the amount of packets transferred

	Delay [ms]	Bandwith [kbps]	Packet loss [%]
MEO	150	492	0
4G/LEO	40	5120	2
VSAT	540	1024	0

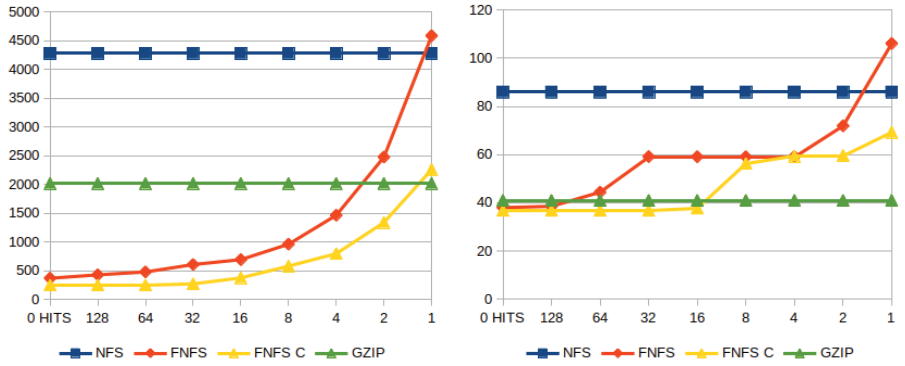
Table 1: Summary of the Evaluated Configurations

(a) and the total tranfer time (b) on a MEO satellite link. For a hit rate lower than 1/3, the amount of packets is reduced, even more if compression is used (1/2). The total transfer time is greater than compressing the file with gzip, only for a hit rate lower than 1/64 w/o compression and lower than 1/16 with compression. Only when there are no hits, the transfer with the standard NFS protocols is better than FNFS. The amount of packets transferred (Fig. 4(a)) for 4G/LEO links is lower for FNFS when the hit rate is lower than 1/3 without compression and lower than 1 when FNFS uses compression. The total transfer time (b) is lower for FNFS only for a hit rate of lower 1/64 w/o compression and 1/16 with compression against zipping the file.

Fig. 5 shows the amount of packets transferred (a) and transfer time (b) on a VSAT link. For a hit rate lower than 1/4, the amount of packets is reduced, even more if compression is used (1/2).

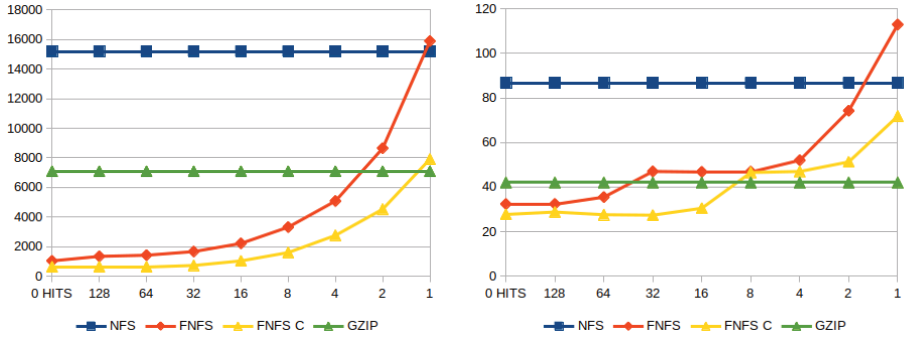
Finally, the transfer time for a VSAT link results better when the file is previously compressed and FNFS has not appreciable advantages over NFS.

As shown in figures 3-5, FNFS demonstrated to reduce the amount of packets transmitted and, in some cases the total transfer time with the exception of a hit rate of 1. In this case, NFS performs better due to the extra work (compression/encryption) made by FNFS. Even though these few cases, FNFS is appropriated to reach the proposed goals.



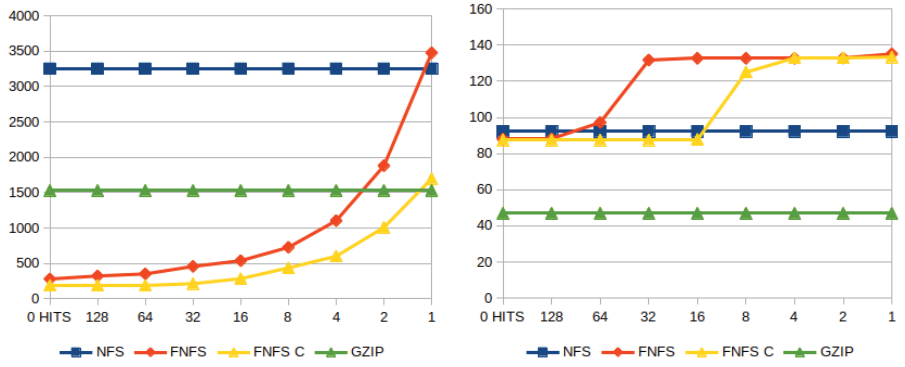
(a) Amount of packets transferred (b) Transfer time

Fig. 3: Results for MEO satellite.



(a) Amount of packets transferred (b) Transfer time

Fig. 4: Results for 4G/LEO.



(a) Amount of packets transferred (b) Transfer time

Fig. 5: Results for VSAT.

5 Conclusions and Future Works

This work proposes to apply the *"filtering useless data at the source"* approach on applications which read a large amount of raw data from a remote server. The filtering rules are specified outside the application and are sent to the server which applies them, afterwards it sends only useful records and encoded useless records to the client. The Client rebuilds useless records with dummy data which will be discarded by the application. This approach reduces network usage, cache buffer utilization, transmission costs in per-packet leased links and battery usage on remote autonomous IoT devices.

To evaluate the proposed approach, popular user-space NFS Client and NFS Server were modified to include filtering capabilities resulting in FNFS. FNFS was designed to keep compatibility with unmodified NFS versions. Moreover, applications that use the protocol don't have to be modified to work with it.

On the other hand, different environment benchmarks reveal a better use of bandwidth, and in some cases lower transfer time. Also, as a consequence of data compression, it reduces the amount of packet transferred. These features translate into buffer saving, smaller numbers of packets to be processed by client and server kernels and fastest communications. In addition, it also helps to keep battery energy when talking about IoT devices, since they usually have limited battery supply. Finally, it may also contribute to decrease communication costs due to decreasing packet transmission for those services whose cost is proportional to the data transferred.

Currently, FNFS is a prototype which can be improved to be used in production environments and only supports a single preloaded filter. Future versions should allow dynamic and multiple filters configurations and data transfer encryption.

References

1. User-space NFSv3 Server, https://unfs3.github.io/
2. HSFS - an NFS client via FUSE, https://github.com/openunix/hsfs
3. J. Kreyig, H. Schukat, H.Ch. Zeider, H. Diel, H. Weber, An intelligent disk controller - A processor system for file management and querry functions, Micropro- cessing and Microprogramming, Volume 25, Issues 1-5, Pages 55-60, ISSN 0165-6074, 1989.
4. ZLIB, A Massively Spiffy Yet Delicately Unobtrusive Compression Library-http://www.zlib.net/.
5. Bruce R. Elbert, The Satellite Communication Applications Handbook, Artech House Space Applications Series, ISBN 1580534902, 2003. Pag. 327 and 406.
6. Satellite Internet access. https://en.wikipedia.org/wiki/Satellite_Internet_access
7. N. J. Upadhyay, Analyzing VoIP for 3G / 4G Wireless Networks, 2009.