A NAS Integrated File System for On-site IoT Data Storage

Yuki Okamoto[1] Kenichi Arai[2] Toru Kobayashi[2] Takuya Fujihashi[1] Takashi Watanabe[1] Shunsuke Saruwatari[1]
[1]Graduate School of Information Science and Technology, Osaka University, Japan
[2]Graduate School of Engineering, Nagasaki University, Japan

***Abstract*—In the case of IoT data acquisition, long-term storage of large amounts of sensor data is a significant challenge. When a cloud service is used, fixed costs such as a communication line for uploading and a monthly fee are unavoidable. In addition, the use of large-capacity storage servers incurs high initial installation costs. In this paper, we propose a Network Attached Storage (NAS) integrated file system called the "Sensor Data File System (SDFS)", which has advantages of being on-site, low-cost, and highly scalable. We developed the SDFS using FUSE, which is an interface for implementing file systems in user-space. In this work, we compared SDFS with existing integrated storage modalities such as aufs and unionfs-fuse. The results indicate that SDFS achieves an equivalent throughput compared with existing file systems in terms of read/write performance. SDFS not only allows multiple NAS to be treated as a single file system but also has functions that facilitate the ease of the addition of storage.**

***Index Terms*—IoT, virtual file system, storage, union mount**

I. Introduction

With the advent of computing devices such as Arduino and Raspberry Pi as well as communication technologies such as BLE (Bluetooth Low Energy) and LPWA (Low Power, Wide Area), the Internet of Things (IoT) is developing rapidly. IoT applications are being introduced in various fields, including industrial manufacturing, agriculture, architecture, civil engineering, energy, logistics, commerce, and education. The success of IoT systems in these diverse applications depends on the ability to build and operate IoT services in real-time. For example, Asahi Iron Works in Aichi, Japan, has built a factory line monitoring system by combining electrical components purchased in Akihabara. In the process, they succeeded in reducing investment by 400 million yen and labor costs by 100 million yen [1].

In the case of IoT data acquisition, long-term storage of large amounts of sensor data is a significant challenge. To address this problem, a low-cost, highly scalable storage system is desirable. When a cloud service is used for the storage of data, fixed costs such as a communication line for uploading and a monthly fee are unavoidable. In addition, large-capacity storage servers have high initial installation costs. Although an on-site IoT system is gradually extended based on the minimum requirements, the use of existing storage systems is not practical. The challenges associated with existing storage systems are discussed in Section II.

In this paper, we propose on a Network Attached Storage (NAS) integrated file system called the "Sensor Data File System (SDFS)", which is on-site, low-cost, and highly scalable. SDFS effectively stores and manages large amounts of sensor data. In this work, we compared SDFS with the existing integrated storages such as aufs and unionfs-fuse. The results indicated that SDFS achieves the equivalent throughput of existing file systems in terms of read/write performance. In SDFS, multiple NAS can be handled as one file system and users can easily add new NAS.

The remainder of this paper is organized as follows. In section 2, we discuss the issues associated with the sensor data storage. Section 3 describes the overview and implementation of the proposed file system, SDFS. In section 4, we evaluate the performance of SDFS and existing file systems. Section 5 introduces related work, and Section 6 summarizes the main findings of this paper.

II. Issues associated with the sensor data storage

With the advent of hardware platforms and communication standards, it has become easier to install sensors and to collect data. For example, we are currently developing a Gunkanjima monitoring system to facilitate building structure analysis based on the acquisition of video and acceleration data of building structures that are collapsing on the uninhabited island [2]. Since 2016, approximately 8 TB of sensor data has already been collected. Storage is a significant challenge when collecting sensor data that is growing over time.

Typically, storage requirements are as follows:

1) Easy expansion and management with low cost
2) Storing data in time-series
3) Existing software can be used

First, easy expansion and management with low cost are important aspects due to the critical importance of the sensor data. Sensor data continues to store with the initiation of data acquisition. In addition, given the recent advancements in data analysis technology, it is desirable to store as much previous data as possible. For example, as mentioned earlier, the Gunkanjima monitoring system has stored 8 TB of data, with an increase of approximately 10 GB per day. The current monitoring system in Gunkanjima stores sensor data on multiple NAS. When the capacity of the NAS is insufficient, it must be switched by manually rewriting the program. In addition, as of this year, we have initiated research on the extraction of the collapse location of Gunkanjima by using acquired image data. As the types and amounts of stored sensor data increase, new demands arise.

Second, considering the characteristics of the sensor data, i.e., the sensor data is associated with time, storing data in time-series is a key requirement. During the extraction of sensor data from a large amount of data for a period, referring to multiple storages individually is inefficient for data utilization. Ideally, close time-series data should be stored in the same storage.

The third requirement is important because existing standard file management software is used for IoT systems such as the Gunkanjima monitering system. Some file operation instructions such as ls, cp, mv, rm, and rsync can still be used for data management.

TABLE I
EXAMPLE OF MONTHLY USAGE AMOUNTS FOR CLOUD STORAGE

Cloud storage	Monthly usage fee [$]
Microsoft Azure Files	480.0
Dropbox	285.05
Amazon S3	200.0
IBM Cloud Object Storage (Cold Vault)	72.0

As a means of storing sensor data, it is conceivable to use a cloud-based storage service. For example, cloud storage such as Amazon S3 is advantageous with respect to using the acquired sensor data in multiple locations. However, it is expensive to store a large amount of data. Table I shows the monthly cost of usage for major cloud storage services. The table provides a summary of the cost associated with the upload of 300 GB of data per month for 8 TB of storage. The usage cost may be higher than the stipulated amount because a transfer fee for processes such as downloading is typically added. In addition, data stores over time and as such, storage charges must continue to be paid as fixed costs. From the perspective of creating new services based on sensor data, systems that require a high level of financial resources for continuous operation should be avoided. A large-capacity storage server is one solution to the problem of the store of large amounts of sensor data. However, storage servers are generally expensive. In addition, there are inappropriate elements associated with the increase of sensor data, such as the upper limit of the amount of data that can be stored.

III. PROPOSAL: SENSOR DATA FILE SYSTEM

A. Overview of SDFS

Based on the discussion in Section II, we consider building a Virtual File System (VFS) that can be considered as a single storage system by integrating multiple NAS. Fig.1 shows the details of access to a file when multiple NAS are integrated. As shown in Fig.1, to manipulate file.dat, the VFS must access each NAS to confirm the existence of the file. When accessing a file, it is not possible to determine which NAS stores the file, therefore, it is necessary to check the existence of the file/directory for each NAS. As the amount of processed data increases, the VFS performs more checks for the existence of files. Therefore, when integrating multiple NAS, the main problem is an increase in the overhead of the VFS.

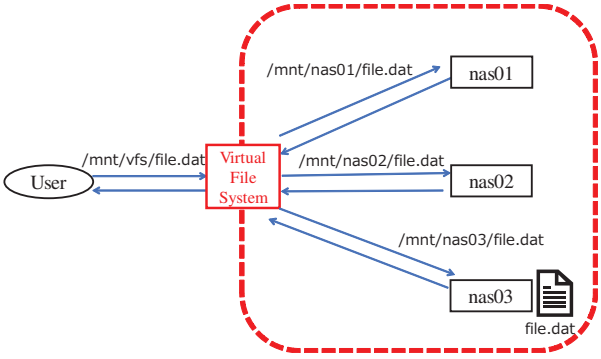

Fig. 1. Access to files in multiple NAS integrated VFS

Based on this discussion, we propose the "Sensor Data File System (SDFS)", a user-space file system for IoT data that satisfies the three requirements described in Section 2. SDFS has the following five features for the efficient management of time-series IoT data.

1) Multiple NAS can be handled as one file system.
2) Users can add a new NAS simply by rewriting the configuration file.
3) When the remaining capacity of the NAS decreases, it is automatically saved on another NAS.
4) Files with similar dates are saved on the same NAS.
5) Users can perform normal operations on one of the integrated NAS.

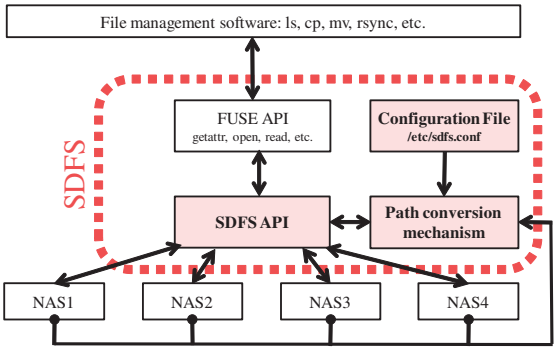

Fig. 2. Overview of SDFS

In SDFS, a format for storing sensor data is standardized as an SDFS path. Specifically, when SDFS is mounted on /sdfs, it is standardized in the following format.

```
/sdfs/[sensor_type]/[year]/[month]/[day]/[name]
```

For example, if the acceleration sensor data on September 10th, 2019 is acc.csv, the SDFS path is following.

```
/sdfs/acc/2019/09/10/acc.csv
```

Figure 2 shows an overview of SDFS when four NAS are integrated. In SDFS, existing file management software such as cp, mv, and rsync can be used. They are used to manage the SDFS path. SDFS is implemented on Linux using FUSE [6], [7]. FUSE is an interface for implementing file systems in user-space. Many file systems have been developed using FUSE for applications such as verification prototypes (e.g., GlusterFS [3], zfs-fuse [4], SafeFS [5]). In the implementation of a file system using FUSE, unique functions can be realized by overriding the hook function corresponding to the system call related to each file operation. When a file management software performs an operation on a file or directory in SDFS, the SDFS API is called via the FUSE API. The SDFS API processes requests for files and directories received from applications in cooperation with the path conversion mechanism. This mechanism converts between SDFS paths and actual paths. It provides the actual path to the NAS to the SDFS API according to the operation target path, the content of the configuration file, the files held by the NAS, and the remaining disk capacity of the NAS. Users edit the configuration file to add a new NAS or select a NAS to store data.

B. Path conversion mechanism: Reading configuration file

Reading the configuration file of the path conversion mechanism is implemented as a thread and is executed periodi-

cally. The thread acquires the NAS mount point information described in the configuration file and checks the remaining capacity of each mounted NAS.

The configuration file is given as a pair of the path pattern of the directory on SDFS, and the mount point of the corresponding NAS. The path pattern is described as an absolute path with the mount point as the root. Users can utilize wildcards. SDFS is mounted on /sdfs, and three NAS are mounted as /mnt/nas01, /mnt/nas02, and /mnt/nas03. At this point, if the configuration file /etc/sdfs.conf contains the following, the files are saved in the order they were written until the capacity is exceeded.

```
/␣/mnt/nas01
/␣/mnt/nas02
/␣/mnt/nas03
```

In SDFS, users can select the NAS to save according to the type, year, and month of the acquired sensor data. If the setting file is described as follows, sensor data from January to September 2017 will be saved in /mnt/nas01. In addition, sensor data from October to December 2017 and 2018 will be saved to /mnt/nas02, and sensor data for 2019 will be saved to /mnt/nas03.

```
/*/2017␣/mnt/nas01
/*/2017/10␣/mnt/nas02
/*/2017/11␣/mnt/nas02
/*/2017/12␣/mnt/nas02
/*/2018␣/mnt/nas02
/*/2019␣/mnt/nas03
```

C. Path conversion mechanism: Checking existence of the file

Reading and writing requirements to satisfy the feature of SDFS are realized by path rewriting of the path conversion mechanism. To check for the existence of files that need to be read and written, the actual path of each NAS must be specified and accessed over the network. In Section III-A, we discussed that the process of integrating multiple NAS in the virtual file system may lead to overhead.

A possible solution is to implement a negative cache to the file existence checking function of SDFS. In this process, information is cached to indicate that the file does not exist. For example, if it is cached that the directory "/mov_b30SS/2019/11" does not exist in the NAS, it can be returned that there is no file for any such request under this directory. In this case, files such as "/mov_b30SS/2019/11/14/building_001.mov" hit the negative cache in the upper hierarchy, therefore, the increase in overhead can be limited. If the file information on the NAS can be returned in the cache without accessing the network for reference when checking for the existence of a file, the overhead can be reduced.

Algorithm 1 shows the file existence checking function in the path conversion mechanism of SDFS, and Tab.II shows the variables and functions used in Algorithm 1.

D. SDFS API: Reading

Reading in SDFS can be performed as if the files distributed in multiple NAS are in one directory. For example, even if NAS1, NAS2, and NAS3 have different files in the directory on September 10, 2019, SDFS can access them as if they exist in three directories.

Algorithm 1 File existence checking function

Require: p
Ensure: return **true** if p exists, return **false** if p not exists
1: $A \leftarrow$split(p, "/")
2: s = ""
3: **for** $i = 1$ **to** size(A) **do**
4: $s \leftarrow$strcat(s, $A[i]$)
5: **if** checkNCache(s) == **true then**
6: **return false**
7: **end if**
8: **end for**
9: s = ""
10: **for** $i = 1$ **to** size(A) **do**
11: $s \leftarrow$strcat(s, $A[i]$)
12: **if** lstat(s) == 1 **then**
13: addNCache(s)
14: **return false**
15: **end if**
16: **end for**
17: **return true**

TABLE II
algorithim VARIABLES AND FUNCTIONS USED IN 1

Variable, Function	Explanation
p	The path to check for existence.
A	A list that stores paths decomposed by hierarchy.
split(s_1, s_2)	A function that returns a string s_1 as an array separated by the string s_2.
strcat(s_1, s_2)	A function that returns a string that combines s_1 and s_2.
checkNCache(s)	A function that checks whether s exists in the negative cache.
lstat(s)	A function to check the status of file s. Returns 0 if it exists, 1 otherwise.
addNCache(s)	A function to register the string s in the negative cache.

Access to the file is read in the order of the longest prefix that matches the configuration file. SDFS reads from the NAS that has the longest prefix match with the path pattern described in the configuration file. At this time, the existence of the file is checked, and reading is performed only when the file exists. If there are multiple NAS with prefix matching of the same length, the existence of the files is checked in the order described in the configuration file. Even if the same file with the same path exists in different NAS, only the file found the earliest using the aforementioned method is read.

E. SDFS API: Writing

The procedure for writing to SDFS consists of three stages. The first stage is the confirmation of the existence of the file to be written. The existence of the target file is confirmed using the same procedure as that used for reading. If the file to be written exists in SDFS, the associated NAS is selected as the write destination and it is overwritten. Otherwise, we proceed to the second stage. If a file with the same name exists in the same path on multiple NAS, the file on the NAS with the highest order listed in the configuration file is overwritten.

In the second stage, matching of the path pattern in the configuration file is performed. The path of the file to be written matches the path pattern described in the configuration file and is written to the remaining NAS with a large capacity. When the remaining capacity of the NAS matched with the path pattern is insufficient, we proceed to the third stage. If there are multiple matching path patterns, the path pattern with the longest prefix match is given priority. If there are multiple

prefix patterns with the same length, the NAS written at the top of the configuration file takes precedence.

In the third stage, the remaining capacity is sufficient and the NAS listed at the top of the configuration file is selected. Saving in the order of the setting files allows sensor data with a similar date and time to be saved in the same NAS, and the time-series to be maintained as much as possible.

IV. Evaluation

A. Evaluation environment

To confirm the usefulness of SDFS, we evaluated the synchronization performance and read/write throughput performance. In the evaluation, the following three file systems are compared.

1) SDFS
   The file system proposed in this paper. This file system efficiently manages a large amount of sensor data. It is assumed that sensor data is managed by multiple NAS.
2) unionfs-fuse [16]
   A FUSE-based implementation of UnionFS [10], [14], [15] that can transparently overlap multiple directories.
3) aufs [11]
   A file system developed to improve the reliability and performance of UnionFS.

There are Samba [12] and Network File System (NFS) as a sharing method of NAS over the network. Both are protocols that enable remote mounting via a network for locally connected storage. The NAS used for the evaluation was constructed on RAID 1 with two 4TB HDD in Synology DS218j.

B. Comparison with existing filesystem

We compare SDFS with existing filesystem using the following practical evaluation criteria.

1) When the number of NAS is increased, performance does not deteriorate.
2) The file system supports Samba.
3) There is no need to rebuild the kernel. It is important for ease of introduction.
4) NAS can be added without rebuilding the file system. If the file system is stopped every time a new NAS is added, sensor data that continues to increase every minute and every second cannot be handled.
5) Users can select which NAS to write according to sensor data type and date.
6) Write according to the remaining storage capacity.

Table III shows the comparison results based on the above evaluation criteria.

TABLE III
Comparison of SDFS and existing filesystem

	SDFS	unionfs -fuse	aufs
1)	○	○	○
2)	○	○	×
3)	○	○	×
4)	○	×	×
5)	○	△	△
6)	○	×	×

Aufs does not support storage shared by Samba. Therefore, we shared the storage by NFS for the performance evaluation of aufs. SDFS, unionfs-fuse, and aufs file systems all exhibited almost no performance degradation even when the number of NAS to be integrated is increased. SDFS and unionfs-fuse are FUSE-based file systems, rebuilding the kernel is not necessary. However, kernel modules are provided for aufs, and it is necessary to rebuild the kernel to use them. SDFS allows the user to add a new NAS simply by rewriting the configuration file, but unionfs-fuse and aufs require a file system rebuild. In SDFS, users can select the NAS to write according to the type/date of sensor data and the remaining storage capacity. Although unionfs-fuse and aufs can select the NAS to write according to the directory ' s name, the detailed specification as described in section III-B is not possible.

C. Rsync performance

We investigated the performance of rsync, which is a Linux file management software that synchronizes files and directories. When a large amount of data is stored and used by sensors, rsync is used for back up. Experiments were performed while changing the number of NAS units to be integrated from 1 to 4 NAS units. We synchronized 10,000 new files with a file size of 256 B using rsync. Fig.3 shows the experimental results obtained when the NAS is shared with Samba. The vertical axis represents the time when rsync is completed, and the horizontal axis represents the number of integrated NAS. Fig.3 shows that SDFS has better rsync performance compared to unionfs-fuse. However, without the negative cache function, SDFS's performance is significantly reduced as the number of NAS increases. Rsync needs to obtain information about a file at various times, including whether the file exists, whether the directory exists, or whether the file needs to be updated. Therefore, when sharing a NAS with Samba, whether or not a negative cache exists greatly affects the performance of rsync.

Fig.4 shows the experimental results obtained when sharing the NAS with NFS. For NFS sharing, aufs has the best rsync performance. SDFS and unionfs-fuse exhibit almost no difference in performance, unlike sharing with Samba.

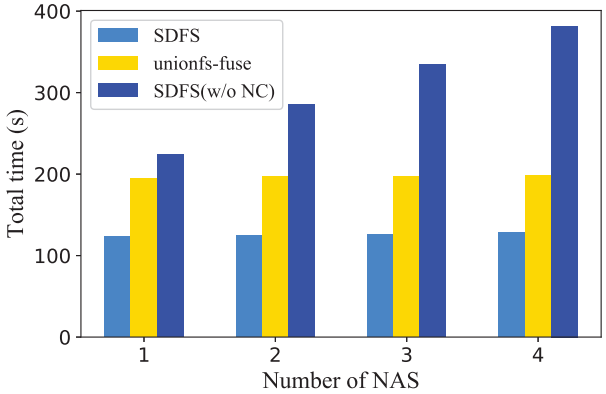

Fig. 3. Rsync performance (Samba)

D. Read/Write performance

We measured read/write throughput. The average write performance was measured by writing the file 10 times and changing the read file size from 1 B to 1GB in 10 times increments. The write throughput is shown in Fig.5 when

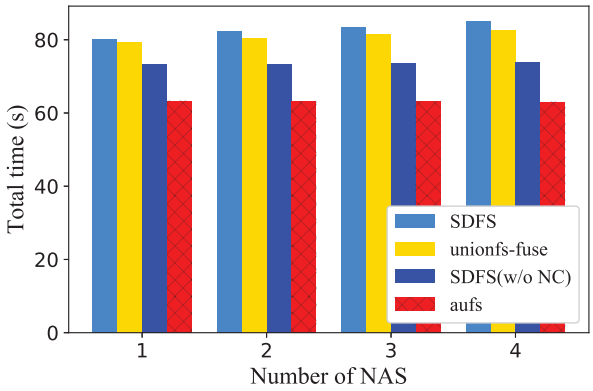

Fig. 4. Rsync performance (NFS)

mounted with Samba and Fig.6 when mounted with NFS. The vertical axis is the throughput (Mbps), and the horizontal axis is the file size (B) when writing.

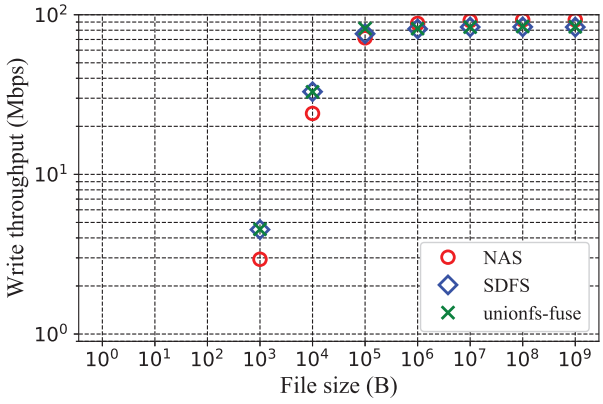

Fig. 5. Write throughput performance (Samba)

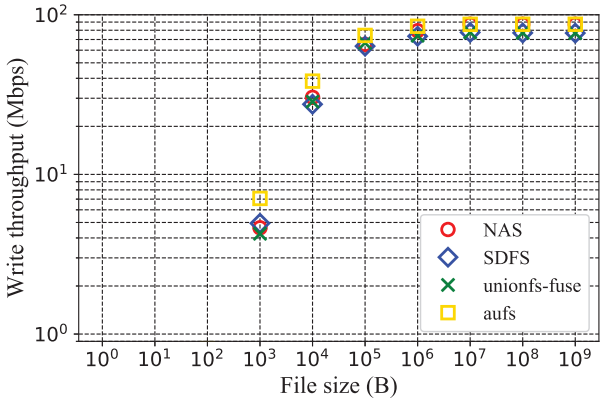

Fig. 6. Write throughput performance (NFS)

The read performance experiment is the same as the write performance. The read throughput is shown in Fig.7 when mounted with Samba and Fig.8 when mounted with NFS. Fig.5-8 shows that SDFS achieves almost the same read/write throughput as existing methods.

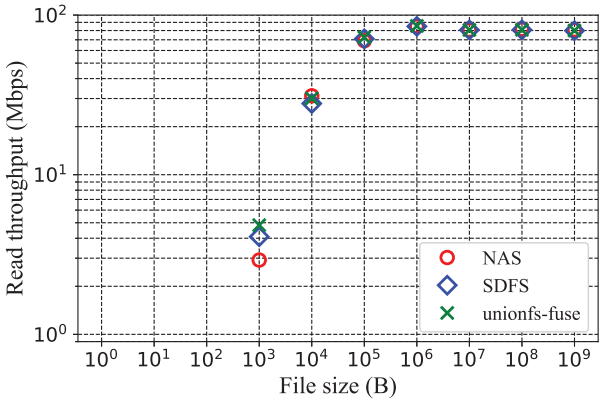

Fig. 7. Read throughput performance (Samba)

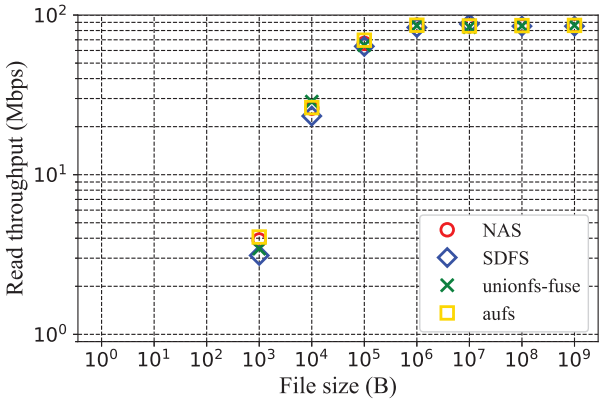

Fig. 8. Read throughput performance (NFS)

E. Negative cache lifetime

In the previous evaluation, the lifetime of the negative cache in the path conversion mechanism was set to 60 s. We examined the effect of negative cache survival on SDFS performance. In this process, four NAS are shared with Samba and the time for completion of rsync is measured while changing the SDFS negative cache lifetime. We synchronized 10,000 files with a file size of 256 B using rsync.

Fig.9 shows the experimental results. The vertical axis represents the time for the completion of rsync, and the horizontal axis represents the lifetime of the negative cache. The negative cache worked effectively except when the cache lifetime was 1 s. When the survival time was more than 10 s, the performance of rsync was almost unchanged.

F. Reading interval of configuration file

In the evaluation thus far, the setting file reading interval in the path conversion mechanism was set to 60 s. The effect of the setting file reading interval on the performance of SDFS is verified. We measured the throughput of writing and reading when four NAS were shared by Samba and the setting file reading interval was changed in increments of 10 from 10^{-4} to

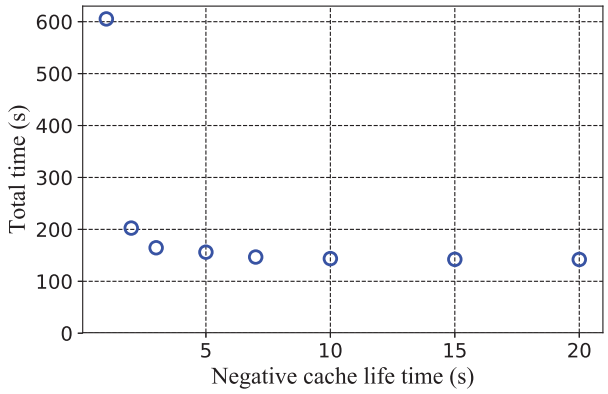

Fig. 9. Rsync performance (Samba)

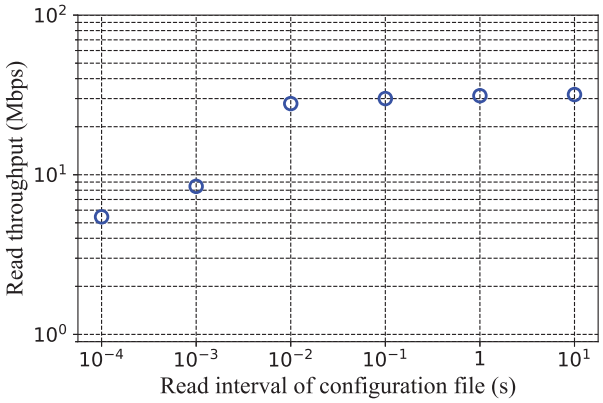

Fig. 11. Read throughput performance (Samba)

10 s. The average throughput was calculated when a file with a size of 10 kB was written and read 10 times. We calculated the average throughput when reading and writing a file with a size of 10 kB 10 times.

Fig.10 shows the write throughput results and Fig.11 shows the read throughput results when the read interval is changed. The vertical axis is the throughput (Mbps), and the horizontal axis is the reading interval of the configuration file (s). If the configuration file read interval is 1 s or longer, read/write throughput performance is not affected.

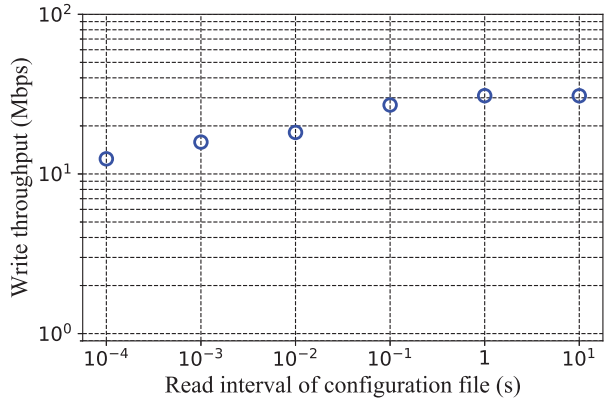

Fig. 10. Write throughput performance (Samba)

V. Related research

Union mount [8] is a technology that transparently superimposes multiple directories such that it appears to contain combined content. An example of using Union Mount is a live Linux distribution that boots from optical media such as CDs and DVDs. In such distributions, writing to optical media is stored in memory. By unifying optical media and memory transparently using Union Mount, it is possible to write onto optical media in a pseudo manner. Typical file systems that use this technology include UnionFS [10], aufs [11], and OverlayFS [13]. There is a demand for file systems that can be stacked, and they have been eagerly developed [9]. Aufs is a file system derived from UnionFS and is used by many services for stability and performance improvement. The SDFS proposed in this paper is considered a kind of Union Mount because it can overlap directories. SDFS is a file system for sensor data that has an advantage when large amounts of IoT data.

VI. Conclusion

In this paper, we discussed the issues associated with the sensor data storage, and proposed SDFS, a multiple NAS integrated file system, which has advantages of being on-site, low-cost, and highly scalable. SDFS not only allows multiple NAS to be treated as a single file system but also has features that facilitate the ease of management of sensor data, and the addition of storage. We confirmed the practicality of SDFS with several evaluation procedures.

References

[1] “IoT success in medium-sized manufacturing firm.” https://news.aperza.jp/ July 2017.
[2] “Gunkanjima monitering project.” https://www-int.ist.osaka-u.ac.jp/battleship/.
[3] “GlusterFS - scalable network filesystem.” https://www.gluster.org/.
[4] “zfs-fuse.” https://github.com/pscedu/zfs-fuse.
[5] Pontes, Rogério and Burihabwa, Dorian and Maia, Francisco and Paulo, João and Schiavoni, Valerio and Felber, Pascal and Mercier, Hugues and Oliveira, Rui. “SafeFS: A Modular Architecture for Secure User-space File Systems: One FUSE to Rule Them All.” Proceedings of the 10th ACM International Systems and Storage Conference. SYSTOR ’17.
[6] Bharath Kumar Reddy Vangoor and Vasily Tarasov and Erez Zadok. “To FUSE or Not to FUSE: Performance of User-Space File Systems.” 15th USENIX Conference on File and Storage Technologies (FAST 17). 2017. feb.
[7] M.Szeredi, “Filesystem in Userspace.” http://fuse.sourceforce.net. 2005.
[8] “Union mounts -a.k.a., writable overlays” http://valerieaurora.org/union/.
[9] Erez Zadok, Rakesh Iyer, Nikolai Joukov and Gopalan Sivathanu and C. P. Wright “On Incremental File System Development.” ACM Transactions on Storage. May 2006. 1553-3077. 161–196.
[10] C. P. Wright and Erez Zadok. “Kernel Korner: Unionfs: Bringing Filesystems Together.” Linux J. December 2004.
[11] “aufs.” http://aufs.sourceforge.net/aufs.html.
[12] “Samba.” https://www.samba.org/.
[13] “overlayfs.” https://github.com/torvalds/linux/tree/master/fs/overlayfs.
[14] C. P. Wright and M. N. Zubair. “Versatility and unix semantics in namespace unificatio.” ACM Transactions on Storage (TOS). 2006.
[15] J. S. Pendry and M. K. McKusick. “VUnion mounts in 4.4BSD-Lite.” In Proceedings of the USENIX Technical Conference on UNIX and Advanced Computing Systems. December 1995. 25-33. USENIX Association.
[16] “unionfs-fuse” https://github.com/rpodgorny/unionfs-fuse.