A Fast and Reliable Real-Time Storage Interface Using File System Watcher

Mohit Shetty[1] Parth Mehta[2] Talha Shaikh[3] Maitry Dave[4] Ms. Vaishali Rane[5]
[1,2,3,4]Student [5]Guide
[1,2,3,4,5]Department of Computer Engineering
[1,2,3,4,5]Thakur Polytechnic, Kandivali, Mumbai, Maharashtra, India

Abstract— Storing, retrieving and manipulating data is one of the most common operation performed by on a normal computer system or a server machine. Data can be stored in different formats such as plain text, binary or custom formats (e.g. JSON, XML, HTML) and in different storages such as main memory (RAM), physical memory (a file system that interfaces a hard drive or any other physical storage) or on some cloud storage. The fastest way to access data on an average computer system would be to access it from its processors' registers or dedicated cache memory, but this might not always be possible and such storages generally have a limited capacity. The next fastest way to access data is the main memory of a system. The average time needed to access the memory can vary from 0.5ns (90% of the time) to 10ns (10% of the time) [1]. However, the data stored in the main memory is temporary and is lost as the power supply gets disconnected. The next option, which is present in most systems is to use a file system that would ensure that the data is permanently stored but the time needed to access it would be comparatively longer. In order to create a fast and reliable storage interface, we'll need to combine the positive points of both the main memory (fastest access time) and file storage system (reliability), with the help of a common file system watcher that keeps track of all the interfaces and this paper exactly discusses a concept based on this idea on an abstract level. The main memory shall be used to access the data and the file system to reliable store it. Whenever the data gets updated on the file system, the file system watcher shall notify all the relevant interfaces and update the main memory. The same notification(s) can then be used to make the storage interface real-time.

Keywords: Storage Interface, Real-Time, Fast, Reliable, File System Watcher, Storage Model, Data Retrieval, Data Storage

I. INTRODUCTION

A. What is a storage interface?

A storage interface can be anything that interfaces a given set of storage(s) or storage interfaces(s) or storage system(s) in order to logically provide a convenient way to provide programmers an abstracted and convenient way to store, manipulate and access data. The storage interface can then be used while either developing programs or a server that could require this concept to create custom storage designs. At minimum, a storage interface should define a way to store and retrieve data. The storage interface described in this paper interfaces mainly two types of storages – the file system and the main memory.

B. What does this paper mean by the terms fast, reliable and real-time (storage interface)?

A fast storage interface is a one that can return the data stored in the memory, in the shortest time possible. A reliable storage interface is a storage interface that makes sure that the data that is once successfully stored, doesn't get lost in case of sudden power loss or any other circumstance and restarting the system would resume things for all the connected reliable components as normal. A real-time storage interface is an interface that is capable of notifying its listeners in case of any change in data.

C. How does the File System Watcher API help?

The File System Watcher API helps by notifying the custom file system watcher of syncing the file system with the main memory and making the interface real-time is played by the File System Watcher API. Whenever there is a change in the file that is being interfaced by an interface, the file system watcher will notify (all the) interface(s) that interface that file in response to which the interface would first update the main memory that it has interfaced and then re-send the notification to all its listeners (that the data has been updated), either locally or via network, which could then either retrieve the newly updated data or simply just ignore the request based on its own state.

II. DESIGN OVERVIEW AND FUNCTIONING

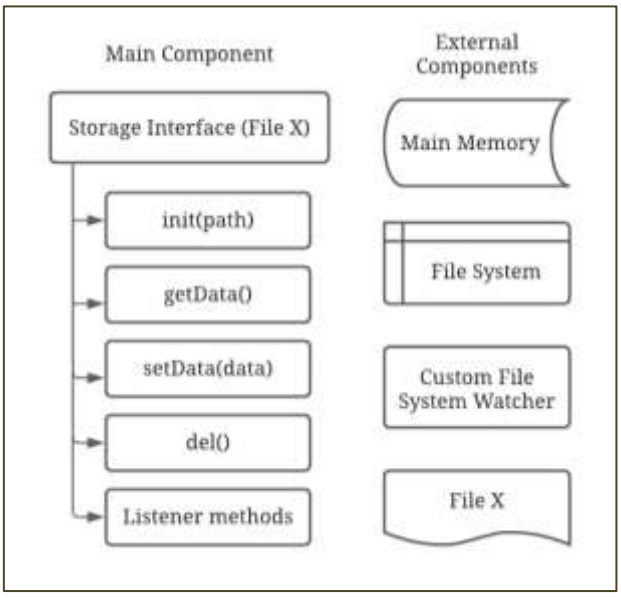

Fig. 1: Design Overview

The main component as mentioned in the title of the paper is the Storage Interface. Each storage interface interfaces a file whose path is provided to it. By default, any storage interface would have four methods defined in it–

1) init(path):

The init method shall be called to initialize the storage interface. The path passed to this method, is that of the file to be interfaced. The method gets the data stored in the interfaced file into the memory and registers itself to the custom file system watcher to get notified about the changes. The method is called implicitly while creating the object and the user. If the file isn't present, an exception shall be thrown/raised for the caller or the object creator to handle. The path given to this method should be stored by the interface for future use (e.g. the setData method).

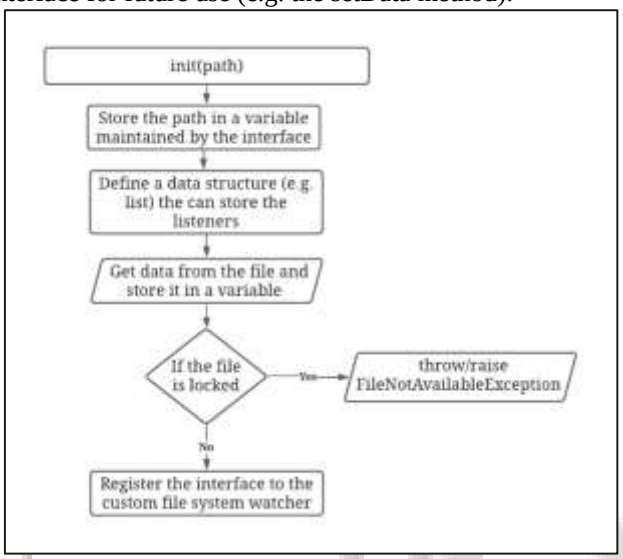

Fig. 2: An example flow that initializes a storage interface

2) getData():

The getData method returns the data stored in the main memory (in high level programming languages as a private variable). It by default does not accept any parameter, however named/optional arguments can be added later while extending this concept further.

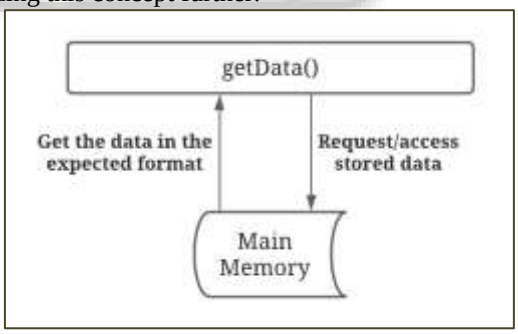

Fig. 3: A diagram that represents the functioning of the getData method

3) setData(data):

The setData method by default accepts a single parameter, where the first parameter represents the data to be updated in the file (and then implicitly the main memory). The setData method tries to write to the interfaced file. (Note: The file isn't permanently locked by the storage interface) If it fails to write either because the file is locked or because the file isn't available (moved/deleted) at the path given while implicitly calling the method, then two unique exceptions shall be raised/thrown for each case. Once the file is successfully updated/modified, a custom file system watcher which shall implicitly update the main memory (the private variable). How the custom file system watcher updates the main memory has been described in detail, while discussing the external components that the interface uses.

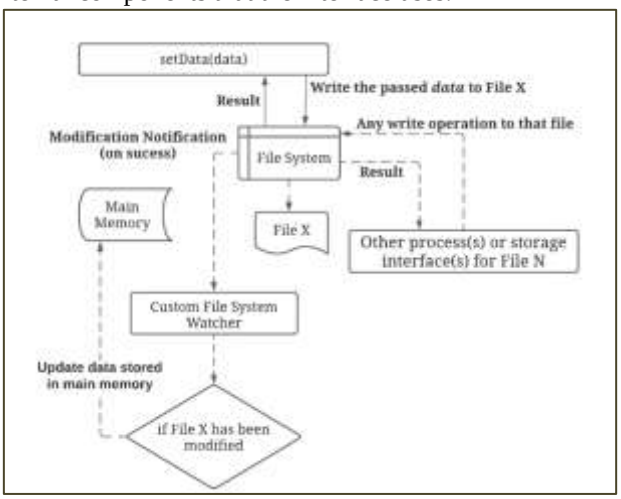

Fig. 4: A diagram that shows how the setData method should work with other components

4) del():

The del method is called when the storage listener is about to get destroyed, either by the programming environment when it's reference is no longer accessible or because the programmer decides to explicitly delete it with the help of the options provided by the language. When this method is called, the main memory used to store the data and the path of the interfaced file is de-allocated and a request to unregister is sent to the custom file system watcher, after which the custom file system watcher would no longer check for the path of that interface (whenever it receives a notification) and the deallocated memory, would no longer be updated or visible for access. This method shall only be called once in the lifetime of a storage interface. Extra option/named parameters shall preferably not be given to this method while later extending the concept to avoid un-necessary complexity.

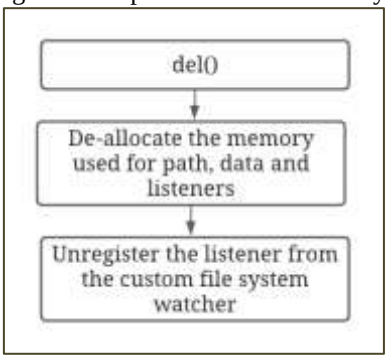

Fig. 5: Flowchart for destructor del()

Apart from the storage interface, there are 5 other external components that the storage interface uses, namely,

1) Main memory:

The main memory is the memory from which the storage interface would always access data from. Based on the language or level at which the storage interface is implemented, the main memory could be a (private) (pointer) variable defined (in a class or interface or namespace) inside a program, from where the data can be obtained via a function/method of the (abstract concept of) interface. This

component is indirectly accessed by the caller when the getData() method called.

2) File System:

The file system is a component provided by almost every operating system. It acts as an abstract for every form of physical storage it supports, and provides different primitives for managing information and primitives that can store those primitives (recursion). Most file system have the concept of files and directories to do most of the work and can have their own ways of addressing a given file, where (the drive/root directory and) every directory until the given file is separated by a forward slash (/) or back slash (\). To support a wider range of operating systems, one can use relative paths separated by forward slashes which would add to the portability of the system that is being designed. The main use of the file system here is to get data whenever the caller, calls this function and set the data, when the program explicitly asks to do so and to notify the File System API on the same.

3) Custom File System Watcher:

The custom file system watcher is a component that uses the File System Watcher API provided by the operating system to get notified whenever any file event takes place within a directory. The custom file system watcher would then iterate through the set of registered interfaces and see if any interface has a path matching to the modified file and update the data in those interfaces and notify the interface to notify its listeners (if any).

4) File X:

File X is the file that is being interfaced by the storage interface. The path of the file is stored by the interface when the init method is called. Whenever the data in this file is modified, the file system watcher API (OS level) gets called, which notifies the Custom File System to call the concerned listener interface methods.

5) Listener methods:

Listeners in this context refer to functions that get called whenever the data of an interface gets updated in any way. In abstract language, they listen to the interface for data changes. A listener can listen to create/modify/delete changes based its type. The custom file system watcher calls the onCreate/onModify/onDelete which in turn selectively calls the listeners attached to that interface.

III. Steps to Implement the Interface

1) Read the paper to get a rough understanding of the idea before you actually start implementing this concept.
2) Decide a common unique directory to specifically store data files for the system you are developing (which would help in avoiding overhead events from unexpected).
3) Create a file for the program/script.
4) Import the required libraries (e.g. watchdog,)
5) Design the custom file system event handler via a class/interface/namespace as defined in the design overview section.
6) Write a few classes to represent every listener uniquely based on its type (helps in using a single queue to store the listeners)
7) Design the main storage interface class as defined in the design overview section.
8) Use the storage interface concept in your program/server (or just simply use an infinite loop while making some minor modifications).
9) Test the code. (To generate the below output we, created a text file called sample.txt, saved it once, deleted it and then saved the main script file the was being executed as a process.)

IV. Extending the Concept Further

This concept in itself can be extended further by supporting other data types or formats of data like JSON or XML and by mapping it to the types offered by the programming language or defined by the user. Alternatively, one could provide better means to conveniently update/manipulate the data by providing new methods that can duplicate the data in the main memory, modify it and update the interfaced file with it or modify the existing setData method's parameters to accept different parameters based on the

V. Making the Interface Real-Time

A. Within a program/process

By default, the interface would be real-time in itself as the interface would accessible from within the program, so the user can directly call getData to fetch the latest data without relying on any listener. However, if the program is GUI based, then one could attach a listener to listen to create/modify/delete events and filter out the events and update the UI with the latest data/state in the function itself. This would conveniently help to ensure that the user is always viewing the latest data.

B. Across different programs/processes (locally)

It is always better to link the programs of the system before executing them in such a way that the programming itself safely manages the data/object sharing part (by implementing a common file system watcher for all the processes). However, in case if that isn't available, then the programmer could directly use file system as the common point to get notified about all the changes while maintaining separate memory for each process (basically creating an interface as and when needed as per the state and requirements of the process) or by using the concept of multiprocessing which may or may not be supported by the language or use a design similar to the network part.

C. Over the network (as a deployed server)

A server that has already been created can use this model to efficiently store and retrieve data. If the client wants to get notified about data changes in real time, then the server can support the (secure) web socket protocol (ws(s)). Whenever the client requests a connection for a specific interface, the server can attach a listener to the interface and notify the client about the changes via the connection. Now it would be, entirely up to the client (application) whether or not it wants to fetch for new data or not. Making the state of the connection synchronous to the state of the storage interface, or maintaining a common unique pool for the storage interface (that can be uniquely identified by their path).

VI. ADVANTAGES

1) The storage interface concept, in its simplest form can be claimed to be the fastest for access for a given computer system as programmatically accessing the memory does not have any visible overhead (except if any by the programming language).
2) Implementing the system from the perspective of a programmer in its simplest in any language where the file system watcher does not have an unfamiliar model is very easy once the core concept of the entire interface is understood well.
3) The reliability provided by the interface is never at the cost of the access time or the guarantee of successfully getting the data as the memory allocated for every new update is separate, until the reference/starting memory location gets assigned to the (pointer) variable. So even if the data of the interface is accessed while the data is getting updated, the current data would be returned until the new data isn't available.
4) The cost and resources needed for the implementation of this system is very low as anyone with basic but accurate knowledge of computer science and how operating systems can implement this system and the chances of malfunctioning are very low, unless the concept is extended or enhanced for any reason.
5) This concept can be used to build better and faster servers, assuming that other factors such as security and hardware reliability are taken into consideration.

VII. PROBLEMS AND COUNTER MEASURES

1) While designing a system with this concept, a lot of things would be needed to taken into consideration such as the constraints of the hardware, number of active users, how often the data updates, in order to avoid unexpected behaviour once the system is deployed. Reading this paper carefully and performing enough research before implementing big systems with this system can help counter these problems. Creating local real-time programs with the concept of interfaces without prior research shouldn't be a problem.
2) The reliability of the system might not be sufficient for all use cases, in which case parallel or a different non-volatile system(s) can be used to store data assuming it has an interface that can be listened to for changes or an interface can be programmed over the existing interface to listen to create/modification/delete based on the local/network device and the storage model being used.
3) Not all file systems might use the same method/primitives to address a given file, so to avoid problems while porting the system from one computer to another with different OS, relative addressing, where each directory reference would be preferably separated with forward slashes. Alternatively, the methods provided by the libraries of a programming being used can be used to join directories and file to form a path or to resolve a relative path into absolute can be used to solve such problems for most of the operating systems that are commonly used.
4) Having too many storage interfaces registered with the custom file system watcher could be end up causing performance issues as the watcher would have to iterate through all the interface would have to iterate through every interface in order to notify the right interfaces, and the interface itself would have to iterate through all its different kinds of listeners that could make things a lot slower. In order to solve this as an issue, a common unique pool of interfaces can be created, where every interface would be uniquely known by the path it interfaces. This would eliminate redundant iteration done by the custom file system watcher. At the level of the interface, for every kind of listener a different queue can be allocated which could reduce the number of iterations but would need more memory for every interface.
5) The idea by default is very plain in itself and hence custom mechanism to retrieve, manipulate or update the data might not be clear in the mind of the programmer. Researching on the format that needs to be implemented and mapping the format to an existing library class or user-defined class might need to by the programmer.

VIII. APPLICATIONS

1) It can be used in local programs to create objects from which we can logically, directly read and write data to.
2) It can be used in GUI-based applications to design dynamic UIs that update when there is a change in data (or the contents of a file).
3) It can be used in servers that quickly need to serve dynamic data as soon as possible.
4) It can be used in applications that need to serve data in real-time to all its clients without any (major) delay or manual request by the user to check for the validity of the data.
5) It can be used to cache commonly accessed files dynamically with the help of a fixed length priority based queue consisting of storage interfaces, interfacing those files.

IX. CONCLUSION

The storage interface described in this paper allows a programmer/system designer to reliably store and quickly access that data in the fastest possible way for a given operating system. By using this interface model, one can conveniently store data for simple programs or GUI-based applications that display dynamic data or different types of server. By writing this paper, we not only described and brainstormed our idea but also related it with its possible real world application.

REFERENCES

[1] The random memory access time data was taken from the highlighted points of the following article: https://www.cs.uaf.edu/2011/spring/cs641/lecture/04_05_modeling.html#:~:text=Average%20Memory%20Access%20Time%20(AMAT)&text=For%20example%2C%20if%20a%20hit,1.4ns%20average%20access%20time.
[2] M. Mesnier, G. R. Ganger and E. Riedel, "Object-based storage," in IEEE Communications Magazine, vol. 41,

no. 8, pp. 84-90, Aug. 2003, doi: 10.1109/MCOM.2003.1222722.

[3] Y. Kang, Jingpei Yang and E. L. Miller, "Object-based SCM: An efficient interface for Storage Class Memories," 2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST), 2011, pp. 1-12, doi: 10.1109/MSST.2011.5937219.

[4] Mary Baker, Satoshi Asami, Etienne Deprit, John Ouseterhout, and Margo Seltzer. 1992. Non-volatile memory for fast, reliable file systems. SIGPLAN Not. 27, 9 (Sept. 1992), 10–22. DOI:https://doi.org/10.1145/143371.143380

[5] A. K. Paul, R. Chard, K. Chard, S. Tuecke, A. R. Butt and I. Foster, "FSMonitor: Scalable File System Monitoring for Arbitrary Storage Systems," 2019 IEEE International Conference on Cluster Computing (CLUSTER), 2019, pp. 1-11, doi: 10.1109/CLUSTER.2019.8891045.

[6] D. Peric, T. Bocek, F. V. Hecht, D. Hausheer and B. Stiller, "The Design and Evaluation of a Distributed Reliable File System," 2009 International Conference on Parallel and Distributed Computing, Applications and Technologies, 2009, pp. 348-353, doi: 10.1109/PDCAT.2009.37.

[7] Gackenheimer C. (2013) Using the File System. In: Node.js Recipes. Apress, Berkeley, CA. https://doi.org/10.1007/978-1-4302-6059-2_3

[8] H. El-Rewini, H. H. Ali and T. Lewis, "Task scheduling in multiprocessing systems," in Computer, vol. 28, no. 12, pp. 27-37, Dec. 1995, doi: 10.1109/2.476197.

[9] H. El-Rewini, H. H. Ali and T. Lewis, "Task scheduling in multiprocessing systems," in Computer, vol. 28, no. 12, pp. 27-37, Dec. 1995, doi: 10.1109/2.476197.

[10] Qveflander, N. (2010). Pushing real time data usingHTML5 Web Sockets (Dissertation). Retrieved from http://urn.kb.se/resolve?urn=urn:nbn:se:umu:diva-36530

[11] Z. Deng and J. W. -. Liu, "Scheduling real-time applications in an open environment," Proceedings Real-Time Systems Symposium, 1997, pp. 308-319, doi: 10.1109/REAL.1997.641292.

[12] SYSTOR '15: Proceedings of the 8th ACM International Systems and Storage ConferenceMay 2015 Article No.: 8 Pages 1 6 https://doi.org/10.1145/2757667.2757670

[13] Rhea, Sean C., Russ Cox, and Alex Pesterev. "Fast, Inexpensive Content-Addressed Storage in Foundation." In USENIX Annual Technical Conference, pp. 143-156. 2008.

