An Agent-Based Model for Analyzing the HPC Input/Output System

Diego Encinas[*][†], Sandra Mendez[‡], Marcelo Naiouf[*], Armando De Giusti[*], Dolores Rexachs[§]and Emilio Luque[§]
[*]Informatics Research Institute LIDI. CIC's Associated Research Center.
Universidad Nacional de La Plata. La Plata, 1900, Argentina.
Email: {dencinas, mnaiouf, degiusti}@lidi.info.unlp.edu.ar
[†]SimHPC-TICAPPS. Universidad Nacional Arturo Jauretche. Florencio Varela, 1888, Argentina.
[‡]Computer Sciences Department. Barcelona Supercomputing Center (BSC). Barcelona, 08034, Spain.
Email: sandra.mendez@bsc.es
[§]Computer Architecture and Operating Systems Department. Universitat Autònoma de Barcelona. Bellaterra, 08193, Spain.
Email: dolores.rexachs@uab.es, emilio.luque@uab.es

Abstract—**High Performance Computing (HPC) applications can spend a significant portion of their execution time doing Input/Output (I/O) operations into files. Improving I/O performance becomes more important for the HPC community, as parallel applications produce more data and use more computing resources. One of the methods used to evaluate and understand the I/O performance behavior of such applications in new I/O systems or for different configurations is using modeling and simulation techniques. In this paper, we present a simulation model of the HPC I/O system by using Agent-Based Modeling and Simulation (ABMS) based on the functionality of the I/O Software Stack. Our proposal is modeled using the concept of white box so that the specific behavior of each of the modules or layers in the system can be observed. The interaction between the layers of the I/O software stack are analyzed by monitoring the internal functions using proprietary parallel file system tools. This allows obtaining the functional and temporal characteristics corresponding to the I/O operations. These characteristics allowed the design and implementation of a representative model of I/O system components. Furthermore, measurements are used to obtain the necessary data sets in the verification, fine-tuning and validation stages. The resulting implementation has shown similar behaviors for measured and simulated values when using the IOR benchmark with various file sizes.**

Keywords–Agent-Based Modelling and Simulation (ABMS); HPC-I/O System; Parallel File System.

I. INTRODUCTION

Many scientific applications benefit considerably from the rapid advance of processor architectures used in the modern High Performance Computing (HPC) systems. However, they can spend a significant portion of their execution time doing Input/Output (I/O) operations into files. Inefficient I/O is one of the main bottleneck for scientific applications in a large-scale HPC environment.

In the HPC field, the I/O strategy recommended is the parallel I/O that is a technique used to access data in one or more storage devices simultaneously from different application processes so as to maximize bandwidth and speed up operations. For its implementation, a parallel file system is required; otherwise the file system would probably process the I/O requests it receives sequentially, and no specific advantages in relation to parallel I/O would be gained. Generally, evaluating the performance offered by a HPC I/O system with different configurations and the same application allows selecting the best settings. However, analyzing application performance can also be a useful before configuring the hardware.

One of the methods used to predict the applications behavior under different configurations of the HPC I/O system is using modeling and simulation techniques. That is, analyzing and designing simulation models based on the parallel I/O architecture allows reducing complexity and fulfilling application requirements in HPC by identifying and evaluating the factors that affect performance. In our previous work [1], we presented a methodology for modeling the HPC system, and validated a first simulation design phase focused on components simulation on the client side. Additionally, the code instrumentation method [2] was used to obtain the calibration parameters for the initial version of the simulator. In this work, we expand our model and description by showing the main agents on both client and server sides in a parallel file system. On the other hand, we apply a more accurate method to obtain calibration parameters using system tools to monitor the internal functions of the file system.

In this article, an HPC I/O system is modeled and implemented using the Agent-Based Modelling and Simulation (ABMS) paradigm. The model was built using the I/O software stack functionality. The different layers were "sensed" by enabling the system's debugging tools. Thus, the necessary data sets were obtained for simulator verification, calibration and validation.

The rest of this paper is organized as follows. Section II briefly describes key I/O concepts, Section III presents the current context of simulation tools for HPC I/O systems, Section IV addresses a functionality analysis for the development of the conceptual model, Section V describes the proposed model, and Section VI describes the computational model of the I/O system. Finally, Section VII presents our conclusion and future work.

II. BACKGROUND

The I/O subsystem in the HPC area consists of two abstraction levels, software and hardware. Usually, the I/O Software includes parallel file system and high level I/O libraries and the I/O hardware refers to servers, storage devices and networks. However, modern HPC I/O system can include more components increasing the complexity of the I/O system.

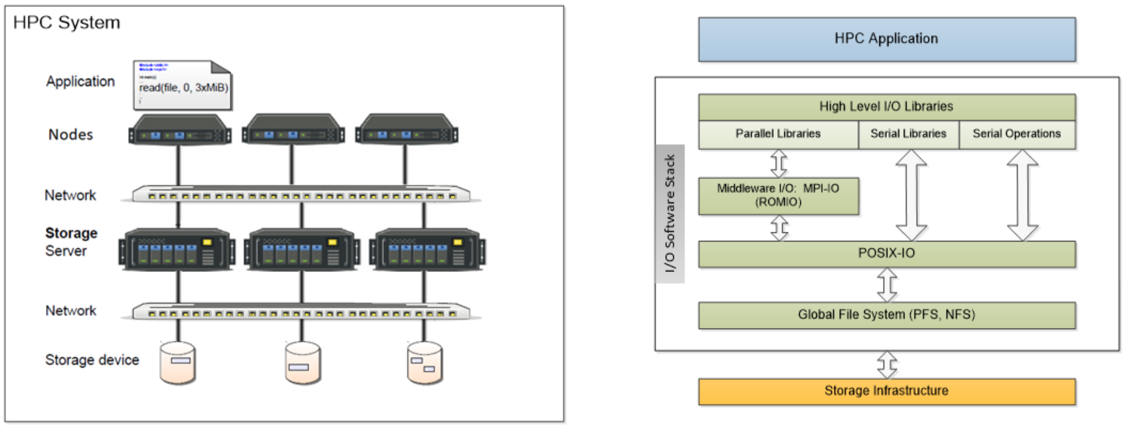


Figure 1. A typical HPC System and the I/O Software Stack

Figure 1 illustrates the structure of the hardware components and the I/O software stack. An I/O operation goes through the software stack from the user application up until it obtains access to the disk from where data are read or on which data are written. Since this parallelism is complex to coordinate and optimize, the implementation of intermediate several layers was designed as a solution.

A. *HPC I/O Strategies*

The most common I/O strategies in HPC are the serial or parallel accesses into files. Serial I/O is carried out by a single process and it is a non-scalable method because operation time grows linearly with the volume of data and even more with the number of processes, since more time will be required to collect all data in a single process [3].

Parallel I/O usually presents two methods or variations of them: `One file per process` and `a single shared file`. In `one file per process`, each process reads/writes data on its own file on disk and no coordination is required among processes. `One single shared file` is more convenient to implement Parallel I/O, where all processes write to the same file on disk, but on different sections of that file. This method requires a shared file system that is accessible to all processes.

There are two ways in which multiple processes can access a shared file: independent access and collective access. In the first case, each process accesses the data directly from the file system without communicating or coordinating with the other processes. In collective access, small and fragmented accesses are combined into larger ones to the file system that helps significantly reduce access times. Our aim is to identify this kind of optimizations to explain the I/O behavior, for this reason, we propose a white box model.

B. *Middleware*

MPI is an interface and communications protocol used to program applications in parallel computers. It is designed to provide basic virtual topology, synchronization, and communication functionalities within a set of processes in an abstract way that is independent from the programming language used to develop the application.

MPI-IO functions work in similar way to those of MPI: writing MPI files is similar to sending MPI messages, and reading MPI files is like receiving MPI messages. MPI-IO also allows reading and writing files in a normal (blocking) mode, as well as asynchronously, to allow performing computation operations while the file on storage device is being read or written on the background. It also supports the concept of collective operations: each process can access MPI files on its own or all together, simultaneously. The second alternative offers greater reading and writing optimizations that can be implemented on several levels. Most of MPI distribution provides MPI-IO functions by using ROMIO [4], which is an implementation of MPI-IO standard and it is used in MPI distributions, such as MPICH, MVAPICH, IBM PE and Intel MPI.

C. *Parallel File Systems*

A parallel file system is a distributed file system that stripes the files data into multiple data servers, connected to storage devices that provide concurrent access to the files through multiple tasks of a parallel application run on a cluster. The main advantages offered by a parallel file system include a global name space, scalability, and the ability to distribute large files through multiple storage nodes in a cluster environment, which makes a file system like this very appropriate for I/O subsystems in HPC. Typically, a parallel file system includes a metadata server with information about the data found on the data servers.

Some systems use a specific server for metadata, while others distribute the functionality of a metadata server through the data servers. Some examples of parallel file systems for high performance computing clusters are IBM Spectrum Scale, Lustre and PVFS2. PVFS offers three interfaces through which PVFS files are accessed: PVFS' native Application Programming Interface (API), Linux kernel's interface, and ROMIO interface.

The underlying complexity of sending requests to all storage nodes and sorting file contents, among other tasks, is handled by PVFS. When a program attempts a reading operation on a file, small sections of the file are read from several storage devices in parallel. This reduces the load on any given disk controller and allows handling a larger number of requests.

D. *Benchmarks*

To evaluate the performance of parallel file system and test different I/O libraries of the I/O software stack, there are different I/O benchmarks. Benchmarks are designed to mimic a specific type of workload in a component or system. One the most accepted I/O benchmark in HPC is IOR [5]. It supports several application I/O patterns and allows configuring them, and it offers access to shared files both independently and collectively. Additionally, IOR offers different execution options for the same algorithm using various parallel programming interfaces, including POSIX, MPI-IO, HDF5 and PNetCDF.

III. STATE OF THE ART

There are several research efforts related to HPC I/O system simulators that focus on storage architecture and some layers of the I/O software stack.

The Simulator Framework for Computer Architectures and Storage Networks (SIMCAN) [6] is aimed at optimizing communications and I/O algorithms. The Parallel I/O Simulator of Hierarchical Data (PIOSimHD) [7] was developed to analyze Message Passing Interface-Input/Output (MPI-I/O) performance. The Co-design of Exascale Storage System (CODES) [8] is a framework developed to evaluate the design of the exascale storage systems. The High-Performance Simulator for Hybrid Parallel I/O and Storage System (HPIS3) [9] models application workload. Lustre Simulator [10] was designed to study the scalability of the Lustre file system.

CODES and HPIS3 are based on Rensselaer's Optimistic Simulation System (ROSS) [11], which is a parallel simulation platform. SIMCAN was developed using OMNET++; PIOSimHD was programmed in Java; and Lustre Simulator, in C++. All the tools mentioned use an event-based simulation paradigm (Discrete Event Simulation, DES). We propose using Agent-Based Modeling and Simulation (ABMS) to develop a simulator that will allow evaluating I/O software stack performance.

The agent paradigm is used in various scientific fields and is of special interest in Artificial Intelligence (AI). It allows successfully solving complex problems compared with other classic techniques [12]. It is a simulation technique that recreates the functionality of different components in a real system by modeling entities known as agents. Basically, an agent is an entity capable of perceiving and acting based on changes in its environment. It can also interact with other agents, executing and coordinating its actions, to achieve goals.

Generally, both paradigms operate in discrete time, but DES is used for low to medium abstraction levels. In ABMS, system behavior is defined at an individual level, and global emergent behavior appears when the communication and interaction activities among the agents in an environment start. In fact, ABMS is easier to modify, since model debugging is usually done locally rather than globally [13].

An advantage of ABMS is that different types of models could be created for each part of the system [14][15]. This is useful because the behaviors of the models differ from each other as they are related to diverse actions like processing, communications and storage. Furthermore, environments could be both software- and hardware-based. ABMS allows implementing different components in a modular and flexible way, affording the possibility of connecting and disconnecting different parts of a complex system for a layer-level analysis.

IV. FUNCTIONALITY ANALYSIS

To define an initial model of the I/O system, system functionality should be fully understood. First, the I/O pattern type to be analyzed was selected, and then the corresponding software stack layers for this model were applied. We have selected the IOR benchmark to evaluate I/O performance in HPC clusters. The analysis was focused on the functionality that was observed for IOR in the data path.

Due to the heterogeneity of the I/O systems and the complexity of the software stack, the analysis was started for MPI-IO layers and the parallel file system. PVFS2 was the file system selected for our tests. At this time, we separated the different components considering the concepts of a parallel file system to allow us using the model with other parallel file systems, such as Lustre in the future.

The IOR benchmark offers the total runtime measurements for their programs, but they do not go into further detail in relation to the different abstraction layers of the parallel I/O system. These layers have to be crossed from the moment the user application sends an I/O request up until the CPU, through its operating system, effectively accesses the file on disk to read or write the data. Therefore, it is important to identify the layer in the software stack that requires more time during an I/O operation.

To follow the data path in the software stack, tracers or monitors can be used, but these operate on different levels of the I/O system. There is no single tool that allows recording the I/O behavior in all levels. However, the parallel file systems typically include logging/debugging methods that allows measuring different parameters on the client and server side.

A. *Monitoring the internal functions of a parallel file system*

The internal functionality of the different components in a HPC I/O system can be identified by: 1) instrumenting the code of the components that are in the data path to perform an I/O operation or 2) using monitoring tools in each level of the software stack. In [1], the code instrumentation in the I/O path was applied to establish what percentage of the total runtime of an I/O operation corresponds to each software stack component. This allows identifying which of them is the most critical one and should be enhanced to improve parallel I/O performance.

The second method requires to monitor the internal behavior of each component of the I/O software stack. As the parallel file system is the I/O software component that is running at client- and server-side, by using its internal logging interfaces, it is possible to identify the internal functionality and its timing for the different component in data path. Some of these tool are Lustre Monitoring Tools (LMT) [16] or Low level Lustre file system configuration utility (lctl) in Lustre [17], or Administration and Monitoring System (AdMon) in BeeGFS [18]. In the case of PVFS2, the options are `gossip` interface and performance counters [19]. In most parallel file systems, these need only to be enabled; they do not require source code re-compilation.

In this paper, we use the PVFS2's `gossip` interface that allows users to specify different levels of logging for the PVFS2 servers. Within the operation principle, `gossip` uses a debugging mask that allows defining which output records are required to print to the log file. Using a global mask, the

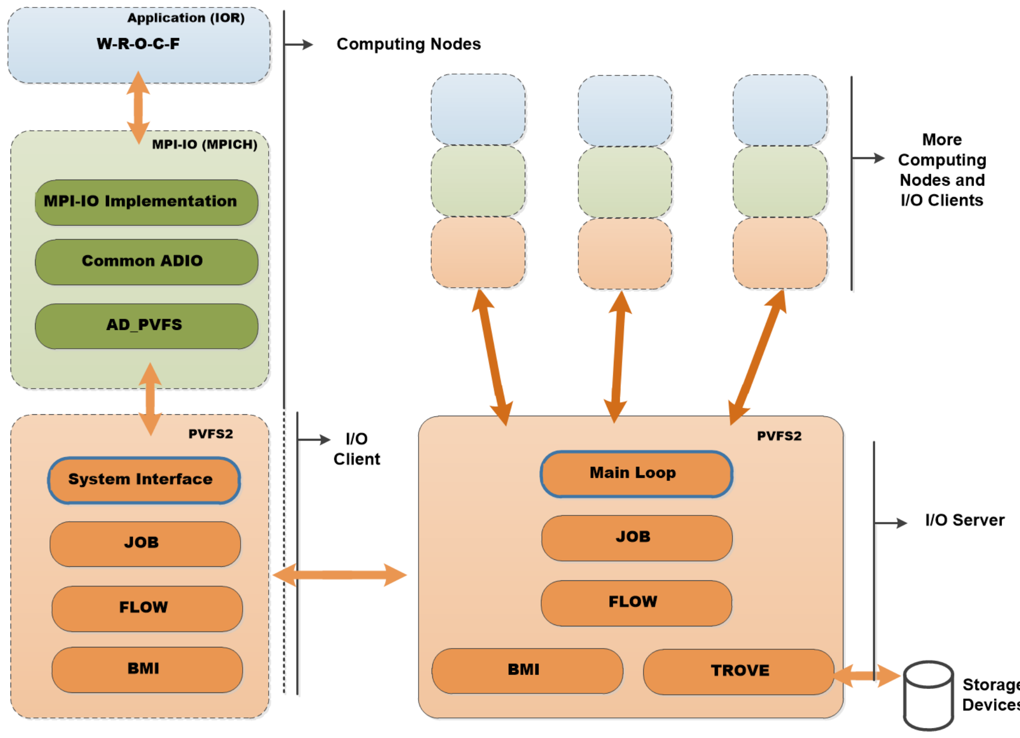


Figure 2. Monitoring in the I/O Software Stack. Left boxes in blue, green and orange represent the layers on compute nodes. The bigger orange box depicts the layers on the I/O Server. Small orange boxes represent the I/O clients, which interact with the metadata and data servers (I/O Servers).

user can specify whether to enable or disable output record sets.

In Figure 2, the different layers of the I/O system can be observed, where the different functions can be measured, both using code instrumentation or the PVFS2's `gossip` interface.

B. *Execution Environment*

One of the problems found in HPC production systems is that the file system cannot be modified/instrumented, and in most cases, the control of the monitoring level of the internal functionality requires root privileges. Therefore, to deploy scenarios to identify the components functionality, we need to have the total control of the HPC cluster and its I/O subsystem. To create the entire I/O software stack with the appropriate monitoring level, we have deployed a small physical HPC cluster with root privileges and a virtual HPC cluster in the Amazon's EC2 platform.

Platforms like Amazon's EC2 offer various types of instances based on the type of service purchased. In [1], a virtual HPC cluster was deployed using the free service and, even though these nodes offer very limited functionality as regards number of CPUs, memory, storage and network; they proved to be adequate to create the necessary environment for the tests executed. Even though this experiment environment allowed obtaining different measurements to be used in the modeling stage, it has already been mentioned that Amazon's EC2 platform service has restrictions.

Unlike execution environment presented in [1], in this work we present the results obtained in a small physical HPC cluster. However, in both scenarios, it was validated that the observed behaviors follow the same trend even though they do not have the same numerical values.

The deployed I/O configuration has five computing and I/O nodes. In this case, an I/O node fulfills the roles of Client, Data Server and MetaData Server for PVFS2. Through the configuration used, the critical functions involved in each layer of the I/O software stack were selected based on their role and execution time. As way of example, Figure 3 shows the functions selected in the System Interface layer on the client-side and Main Loop layer on the server-side.

V. MODELLING THE I/O SYSTEM

To design a model, the basic characteristics of real system behavior must be obtained first [20]. In this case, the interactions between control, data and communications for basic I/O operations were analyzed: open, read and write. Each operation triggers a succession of interactions that, in turn, initiate different functions such as those shown in Figure 3 in each of the layers of the I/O software stack.

A. *System Interactions*

The different interactions between client and server to perform read (**r**) and write (**w**) operations are shown in Figure 4. Once both client and server have been initialized, the System Interface layer starts the **r/w** operation. Since every operation that involves communication with Buffered Messaging Interface (BMI), Flow or Trove is considered a Job, a new operation is indicated to the client's Job layer. The Job layer then sends a message to the Flow layer to start a new transmission flow and send a message to BMI, which immediately establishes communication with the server's BMI. In the server's BMI buffer, the message containing the operation to be performed, the job identifier, the associated flow and a BMI client identifier are added.

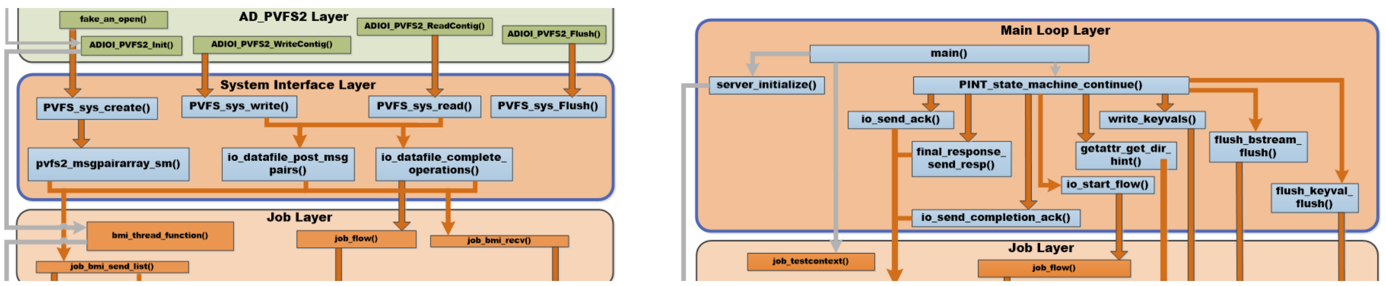


Figure 3. Selected functions of System Interface and Main Loop layers.

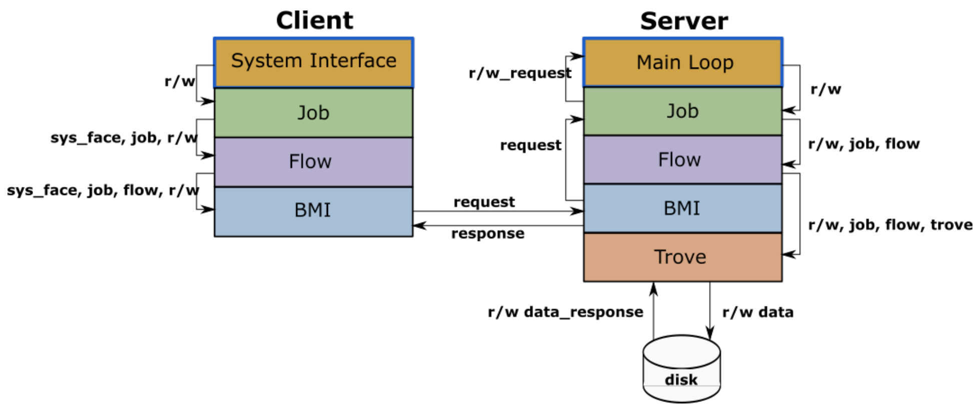


Figure 4. View of the Server-Client Interaction for read (r) and write(w) operations.

On the server-side, once the new operation is detected, the I/O operation is identified and communicated to the Main Loop layer. This layer sends a request to the Job layer to start the new job related to a new flow. Then, a transfer from the Flow layer to disk is carried out. Figure 5 shows how the Flow layer finishes the operation, a response is formally sent to all server layers, and then the server's BMI layer communicates the client's BMI layer that the operation ended.

Figure 4 represents the basic interactions between client and server at the sequential level, but there are other interactions that run in parallel. To carry out an analysis of parallel functionality, the sequence diagram shown in Figure 5 was used. The diagram distinguishes 3 operations: client initializations, server initializations, and the I/O operation itself. As it can be seen, the initializations are run in parallel (highlighted in a blue box for the client and in a green box for the server). Initializations have two purposes: on the one hand, initializing the communications layer on both the client and the server. On the other hand, informing the server that the client is available to establish a communication.

In all interactions, different parameters are sent to each layer in the PVFS2 software stack to identify and carry out the required operation. After the initializations have been performed, the requested I/O operation is executed. The following interactions are sequential and correspond to those mentioned in the description of Figure 4.

B. States Machines

After analyzing each of the layers, a model of the I/O system was developed by implementing state machines and variables that describe each of those states. To that end, state machines were implemented for each of the layers in the system, differentiating their operation both on client- and server-side. The ultimate goal is using these state machines to design the behavior of each of the agents and its interactions with other agents and/or its environment.

The model developed is aimed to reproduce the interaction among the different components and analyzing how the information goes through the different modules or layers, with the possibility of measuring time to approach the real model of the I/O system. Therefore, each layer is modeled based on the execution flow of the functions that are called while processing certain requests, such as opening, closing, reading and writing operations. With the description of each function, the different states of the layers while carrying out those requests were implemented.

Due to the complexity to describe fully the modeling of the I/O software stack, we have selected the System Interface and Main Loop layers to explain in detail the calibration, verification and validation phases. Similar steps were done for the other layers.

The System Interface layer is a client-side interface that allows manipulating the objects in the file system. It launches a number of functions and state machines that process the operation in small steps. In turn, the Main Loop layer is a server layer in charge of controlling whether the operations on lower layers executed by different threads have been completed.

In the context of PVFS, state machines execute a specific function in each of their states. The value returned by this function determines the state that should be adopted. Complex requests can be modeled; they are represented as a sequence of several states. Also, state machines can be nested to model and simplify common subprocess handling. These machines

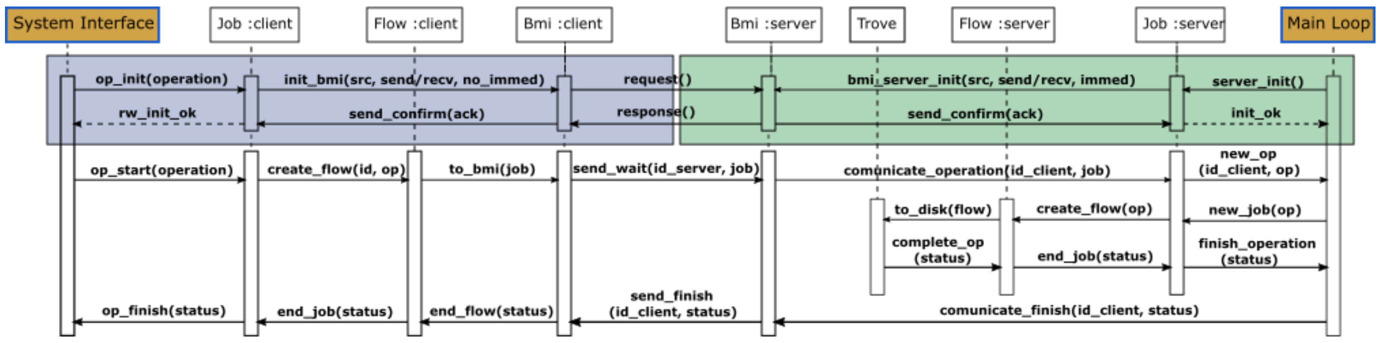


Figure 5. Interaction diagram of System Interface and Main Loop layers for read (r) and write(w) operations.

are used both in clients and servers.

There are several caches on the client side that are part of the System Interface layer and try to minimize the number of requests that the server has to process. The attributes cache (acache) manages metadata, while the name cache (ncache) stores the filename of file system objects and their respective handling number. To prevent caches from storing invalid information, data are set as invalid after a certain time has passed or when the server notifies the client that the object does not exist.

The Main Loop layer accepts four different types of return values related to the invoked operation: completed, deferred, terminated, or failed. It should be noted that the Main Loop layer has one more operation in addition to open, write and read. This is because initialization is an operation in itself, either as a Dataserver or a Metadata server.

C. Functional Model

As shown in Figure 3, the functions in the system interface layer are: `PVFS_sys_create()` to manage the creation of new files in the system, `PVFS_sys_write()` to perform writing operations, `PVFS_sys_read()` to perform reading operations, and `PVFS_sys_flush()` to dump file to data server. The most significant functions of the Main Loop layer include `io_send_ack()`, which returns a negative or positive response to the client; `io_send_completion_ack()`, which reports the completion of an operation that was in progress; and `io_start_flow()`, which initiates a Job to service a Flow depending on the requested operation. Each of these functions has internal variables and state machines that are run to carry out the relevant operations.

Each of these functions has internal variables and state machines that are run to carry out the relevant operations.

To simplify the model, we considered the following in relation to parallelism when handling several instances:

- I/O interfaces: layers `MPI-IO`, `ADIO` and `AD_PVFS` work in a sequential and blocking manner, since they run functions that require synchronization; this means that no instruction is served until the instruction being processed is completed. The calls run on their state machines are blocking;
- PVFS2 parallel file system: the System Interface, Job, Flow, BMI, Main Loop and Trove layers serve other requests and store their instructions in a buffer. Therefore, it allows handling different data flows.

The behavior of each of the agents is described by the state machine, the state transition table and the corresponding state variables. Figure 6 shows part of the state machines developed to model the operation of the System Interface layer, considering the functions and state machines corresponding to each of the three initial operations. As it can be seen, it consists of four agents called System Interface, which is responsible for decoding the instructions that enter the layer; `PVFS_OPEN`, which manages file opening operations; `PVFS_GETATTR`, which carries out searches in the metadata; and `PVFS_RW`, which manages file reading and writing operations.

As way of example, the states of an agent in the System Interface layer in Figure 6 are explained. The agent that manages file opening operations can only have one of five different states (S8 to S12). It will remain in S8 and configure agent `PVFS_GETATTR` if it requests metadata. If the attributes are not found in cache, it will transition to state S9 to wait for them; otherwise, it will transition to state S10. If in state S9, it will wait for a response from agent `PVFS_GETATTR` or it will complete the opening operation by communicating with the server, transitioning to state S10. If the operation cannot be completed, it will transition to state S12 to end.

While in state S10, it will start file creation through a request sent to the JOB layer, transitioning to state S11. Otherwise, it finishes the operation and transitions to state S12. While in state S11, it waits for a response to its file opening request and, if it receives one, it transitions to state S12. Once in state S12, it finishes the operation and sends a response to agent `AD_PVFS`.

Each state of the `PVFS_GETATTR` agent, the same as each of the agents in each layer of the system, has different state variables. These are five per state, and their values depend on their role: ID to identify each process, `DATA_SYSTEM` to indicate permanence in memory or not, `OPERATION` to specify the type of operation, `REQUEST_IN_PROCESS` to indicate if the process has finished or not, and `COD_OPERATION` to add an identifier per traversed layer.

On the other hand, agent `PVFS_RW` manages the write or read requests on client side. In Figure 6, there can be seen in red the functions selected that were used as the basis for the development of each state machine. For example, one of the functions belonging to `pvfs2_msgpairarray_sm()`[21], on which the `PVFS_RW` agent is based, is `io_datafile_post_msgpairs()` that is responsible for managing the data transmissions involved in the creation of files in agent System Interface. These communications occur,

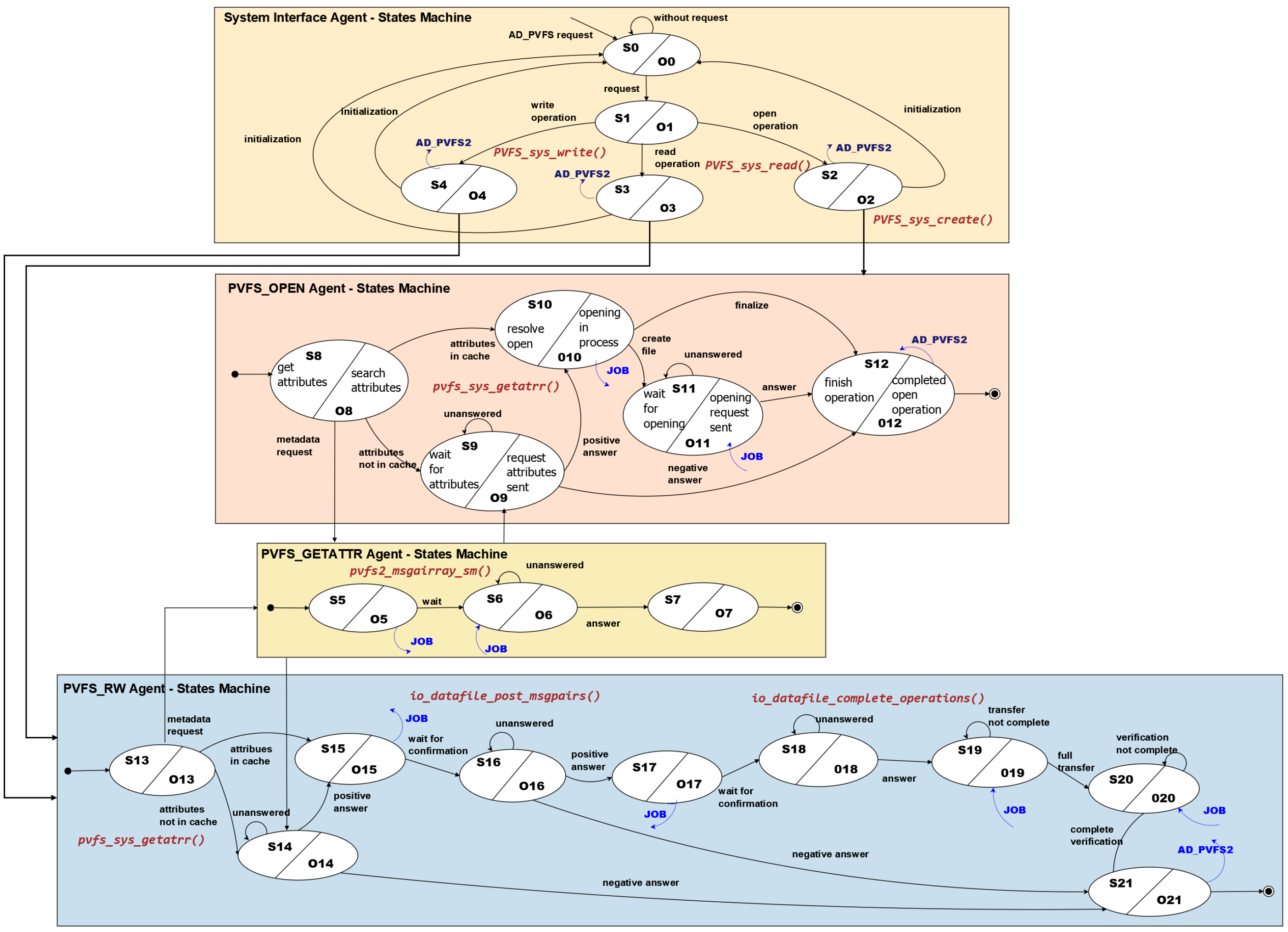

System Interface Agent - States Machine
PVFS_OPEN Agent - States Machine
PVFS_GETATTR Agent - States Machine
PVFS_RW Agent - States Machine
PVFS_sys_write()
PVFS_sys_read()
PVFS_sys_create()
pvfs_sys_getatrr()
pvfs2_msgairray_sm()
io_datafile_post_msgpairs()
io_datafile_complete_operations()

Figure 6. State machines for agents in the system interface layer.

in the case of both a reading or writing, between client and server through the Job and BMI layers.

As for the server's Main Loop layer, Figure 7 shows how it is modeled with 4 agents, namely: `MainLoop`, which handles server initialization and decodes required operations; `MetaData Creation`, which reads metadata from disk immediately; `File Creation`, which writes new files or directories metadata to disk immediately; and `Read/Write`, which is responsible for configuring data transfers to disk and sending acknowledgment signals to the client.

Figure 7 shows the state machines of each agent, with focus on the states of the Read/Write based on the roles corresponding to `pvfs2_io_sm()`), which have been marked in red. As previously mentioned, this agent is in charge of managing the data reading or writing operations requested by the client.

VI. COMPUTATIONAL MODEL OF THE I/O SYSTEM

To develop the simulator, tasks were organized in three groups: 1) obtaining data sets that represent the temporary function of the system, 2) using an ABMS-oriented framework, and 3) validating the tool developed.

A. *Verification and Calibration*

To obtain values for the functional model, we have monitored the selected functions for the IOR benchmark in a HPC physical cluster. The I/O system was configured over on PVFS2 parallel file system and the MPICH distribution. The cluster was composed by five nodes, where each one had three roles: compute node (computing and PVFS2 clients), metadata server and data server (datafiles).

We have selected the IOR [5] benchmark as application and it was configured to run a simple pattern for different file sizes and transfer sizes. IOR was configured as follows:

- 1 GiB === `mpirun -np 5 ./ior -a MPIIO -b 205m -t 205m -F`
- 2 GiB === `mpirun -np 5 ./ior -a MPIIO -b 410m -t 410m -F`

For this setting, each process writes/reads to/from its own file in transfer sizes defined by the `-t` parameter. Due to the block size (`-b`) is equal to the transfer size (`-t`), only one operation is done by each process. The interface selected was MPI-IO for the *one file per process* (`-F`) strategy and independent I/O. The mapping corresponds to one MPI process per compute node.

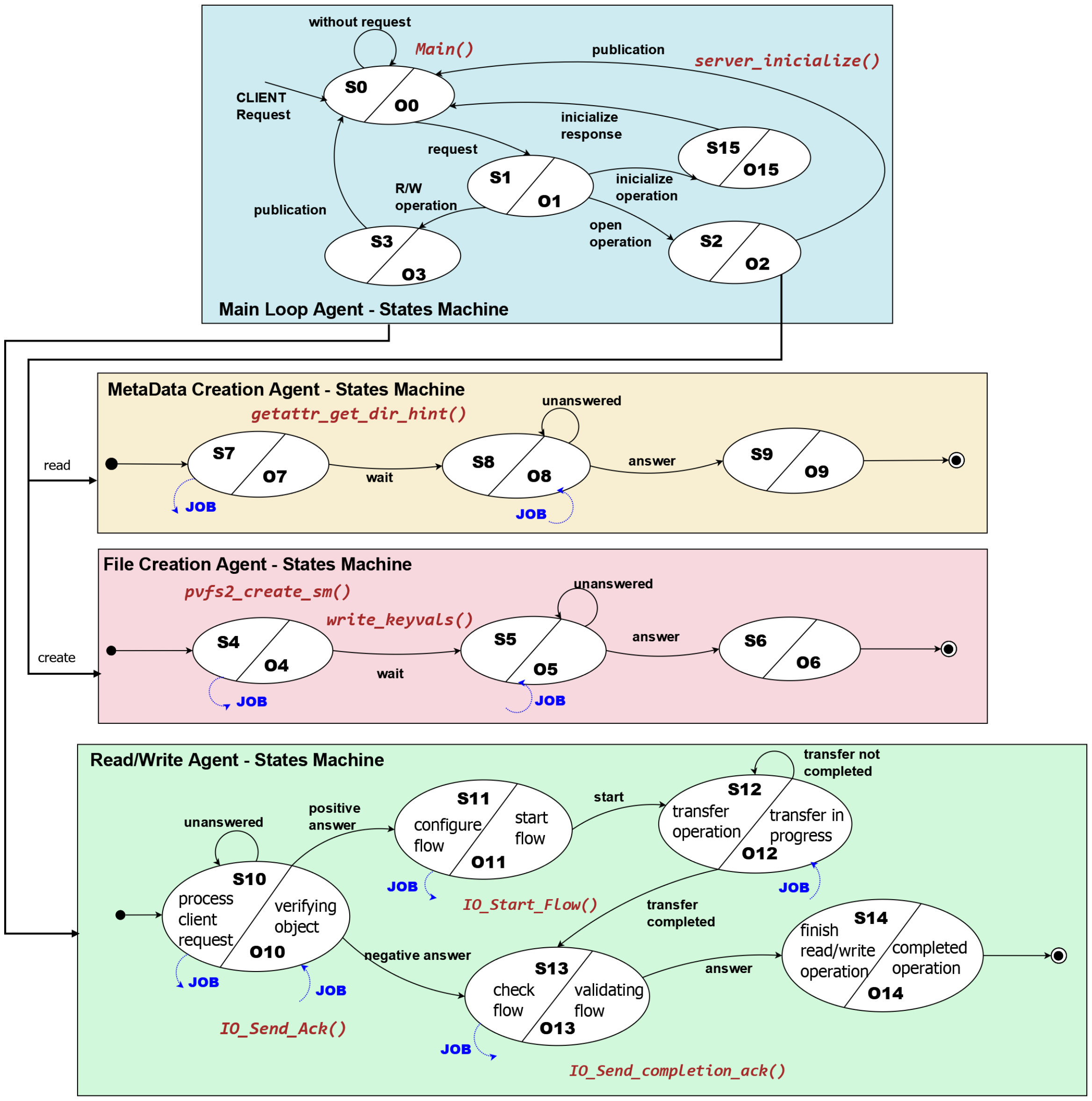


Figure 7. State machines for agents in the main loop layer.

This measurement allows us to classify the monitored metrics in three groups: 1) *data access time* related with the data accesses operations such as write, read, and so on, 2) *control time* that includes verification and configuration of the data structures and 3) *communication time* related with the interaction between the clients and the metadata and data servers.

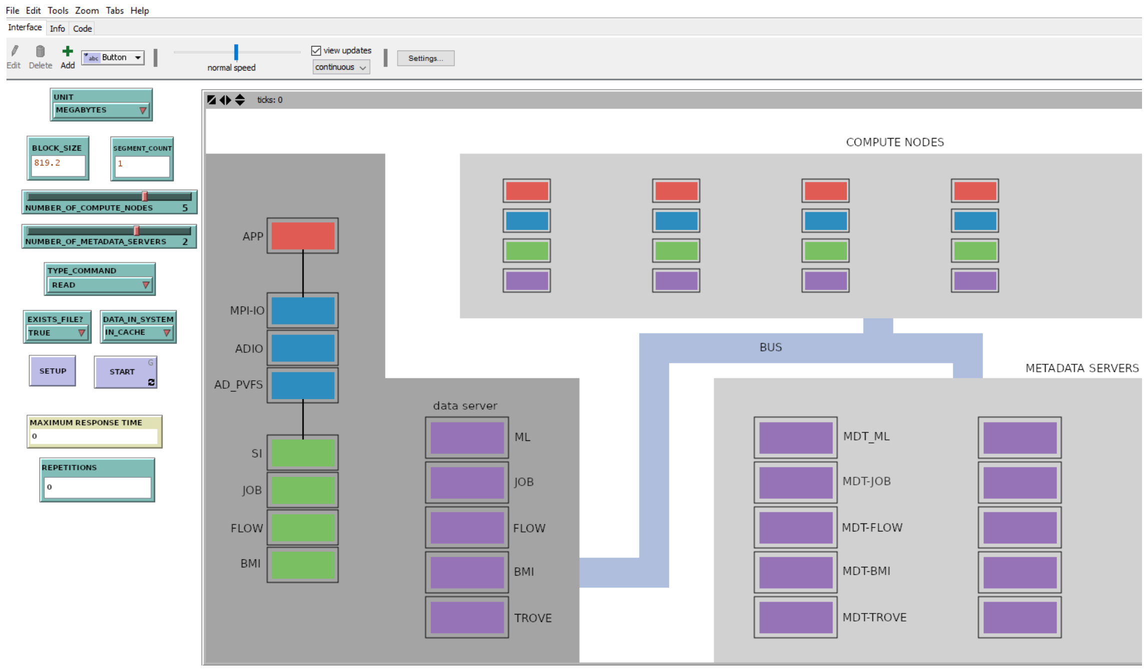
Activating the PVFS2's `gossip` interface the metrics were obtained to apply linear and exponential regressions for the time monitored in different PVFS2's functions. For this analysis, we have selected as dependent variable the execution time and as independent variable the file size, request size is fixed for all the tests. In the case of the system interface layer, we have selected the following equations to represent the time of the functions:

- `PVFS_sys_create()` $= 0.0217x$
- `PVFS_sys_write()` $= 15.183x + 0.0408$
- `PVFS_sys_read()` $= 15.167x + 0.0376$
- `io_datafile_post_msgpairs()`$= 0.002x^3 - 0.0137x^2 + 0.027x - 3 \cdot 10^{-15}$
- `io_datafile_complete_operations()`$= -5.6305 \cdot 10^{-7}x^4 + 5.3594 \cdot 10^{-6}x^3 - 1.7401 \cdot 10^{-5}x^2 + 2.1925 \cdot 10^{-5}x + 7.2760 \cdot 10^{-20}$

The equations representing the time functions of the main loop layer are defined as follows:

- `io_start_flow()`$_{read} = 11.3549x$
- `io_start_flow()`$_{write} = 11.4889x$
- `io_send_ack()` $= 3.1987 \cdot 10^{-6}x^3 - 2.4538 \cdot 10^{-5}x^2 + 5.6331 \cdot 10^{-5}x$
- `io_send_completion_ack()`$= 7.1776 \cdot 10^{-6}x^3 - 5.5622 \cdot 10^{-5}x^2 + 0.00012x$

Where the x variable represents the file size to write or



UNIT: MEGABYTES
BLOCK_SIZE: 819.2
SEGMENT_COUNT: 1
NUMBER_OF_COMPUTE_NODES: 5
NUMBER_OF_METADATA_SERVERS: 2
TYPE_COMMAND: READ
EXISTS_FILE?: TRUE
DATA_IN_SYSTEM: IN_CACHE
MAXIMUM RESPONSE TIME: 0
REPETITIONS: 0

Figure 8. Simulator's user interface in NetLogo

read. The statistical dispersion also depends on the file size and therefore it is calculated by using the same method.

B. Implementation

The simulation model was developed using an ABMS framework called NetLogo. This framework includes a simplified programming language and a graphical interface that allows the user build, observe and use agent-based modeling without understanding complex standard programming language details. This tool is specifically indicated for the simulation of complex systems; it allows giving instructions to many independent agents that are concurrently executed, which is useful to study the connection between individual and collective behavior through agent actions and interactions.

An implementation detail in this simulator is the use of an agent called "data" that can be invoked by other agents. This new agent has two main objectives – the first is to calculate the execution time of a function in terms of file size, since the "data" agent can invoke a set of models, algorithms and functions of system components in NetLogo language. The second objective is to generate the simulator output showing the data associated with the invoking agent. Thus, the name of the invoking agent, the associated function based on its state, and the execution time of the function can be displayed.

The scenario adopted for the experiments is similar to the one used in [1], and it was designed to simulate the exchange of information among computing nodes, I/O nodes and storage nodes considering in each of them the layers discussed in previous sections. The MPI operations that can be served by the application layer are only I/O operations, and this initial implementation only includes open, read, write and close operations. One of the parameters allows toggling between executing only one type of operation or all of them. There is an option for selecting a maximum number of operations, which are distributed among the computation nodes selected.

The number of computation nodes and storage nodes can be configured. Node actions and interactions were fully implemented for the operations mentioned above. There are other parameters that allow selecting the existence of the data in the system before running the simulation, configuring the corresponding layers and preparing the I/O server for this scenario.

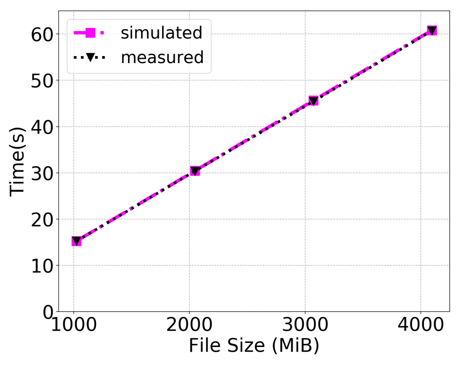
Figure 8 shows the simulator's user interface. The configuration bars that the user has available to set the variables and parameters of the I/O software stack and the scenario to simulate are on the left. Also, the I/O configuration can be made through command line. The center shows the distribution of the I/O system.

C. Validation

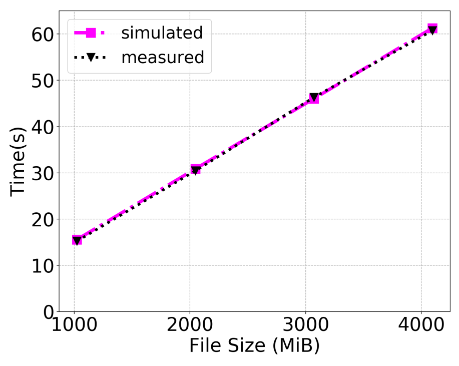
To validate the proposed model, we have configured a physical cluster similar to deployed in the calibration phase (see Section VI-A). The I/O system was deployed by using the PVFS2 parallel file system in a HPC cluster composed by five nodes, where each one was compute node (computing and PVFS2 clients), metadata server and data server (datafiles). PFVS2 filesystem was configured with a stripe size of 64 kiB and a total capacity of 950 GiB. IOR was executed for the following configurations:

- 1 GiB === `mpirun -np 5 ./ior -a MPIIO -b 205m -t 205m -F`
- 2 GiB === `mpirun -np 5 ./ior -a MPIIO -b 410m -t 410m -F`
- 3 GiB === `mpirun -np 5 ./ior -a MPIIO -b 615m -t 615m -F`
- 4 GiB === `mpirun -np 5 ./ior -a MPIIO -b 820m -t 820m -F`

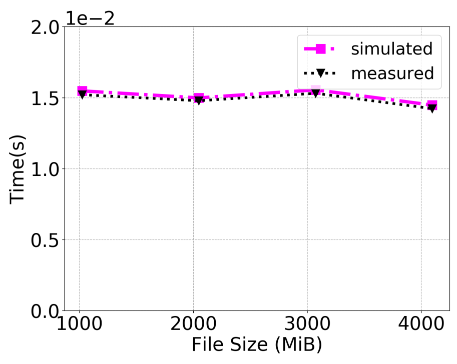
Figure 9 presents the simulated and measured times for the IOR benchmark in the System Interface layer of the PVFS2. As can be seen, the I/O behavior in this layer is dominated



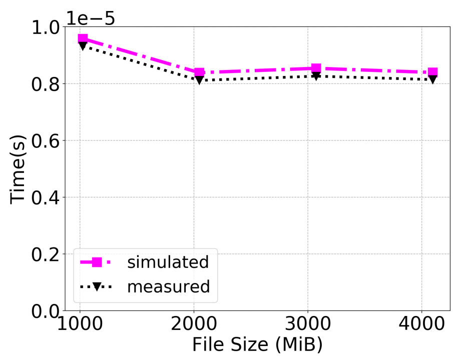
(a) read operations



(b) write operations

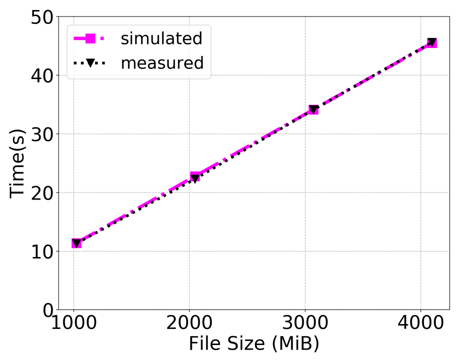


(c) control operations

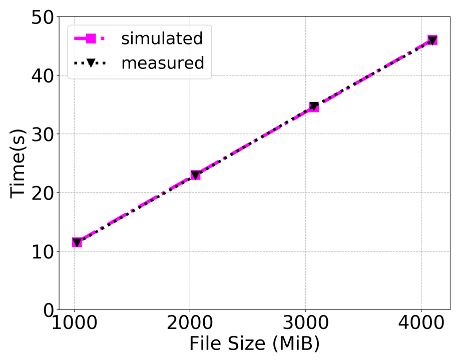


(d) communication operations

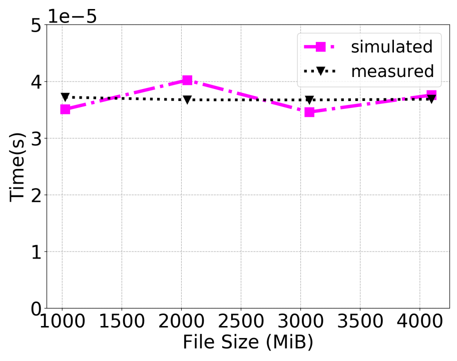
Figure 9. Simulated and Measured time for the system interface layer on the PVFS2's client side



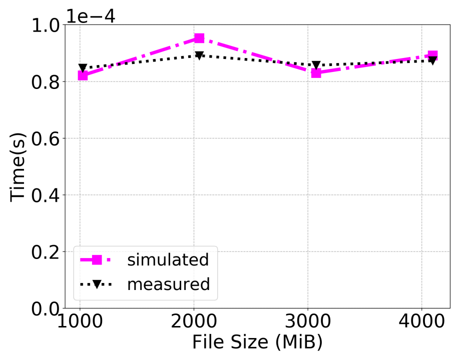
(a) read operations



(b) write operations



(c) control operations



(d) communication operations

Figure 10. Simulated and Measured time for the main layer on the PVFS2's server side

by the access data operations that corresponds to the read and write operations. Timings of control and data access operations are very close for 3 GiB and 4 GiB files, which were not tested in the calibration stage. (Section VI-A). Only in the communication operations can be observed a fixed small gap.

Figure 10 shows simulated and measured results at Main Loop level (server-side). Data access operations present a very similar behavior, but we can see different values for the control and communication operations. This is mainly related with functions and constants that are not adjusting perfectly with the real measurements.

The main reason of the accuracy in the measured and simulation results is the simple I/O pattern and the configuration selected. However, this simple HPC I/O system configuration allows us to show that is it possible to model the I/O system behavior properly by using ABMS. Furthermore, from this model, we can deploy different scenarios for the HPC I/O system, including both hardware and software components.

VII. CONCLUSION

This paper presented a model of HPC I/O system by using ABMS, where agents interact and communicate within the I/O software stack layers. To obtain a more representative time for the calibration functions, the interaction between the software stack layers corresponding to the file system were logging with the `gossip` interface provided by PVFS2. A functional model was defined for the different components of the HPC I/O system by using state machines. The measurement allowed to define equations that represent the temporal behavior for the I/O software stack layers. Furthermore, this was useful for the verification and calibration stages and also for the validation of the simulator developed with the NetLogo modeling environment.

As future work, we will deploy different scenarios for analyzing possible configurations both hardware and I/O software stack. Furthermore, we will evaluate collective operations and other I/O strategies. Additionally, we will extend the model for other parallel file systems, such as Lustre or BeeGFS.

On the other hand, by using the tools for the measurement, we have detected other parameters that can be included in the model and implemented in the simulator, i.e., the data transfer rate (bandwidth) and the input/output operations per second (IOPs).

ACKNOWLEDGMENT

This research has been supported by the Agencia Estatal de Investigación (AEI), Spain and the Fondo Europeo de Desarrollo Regional (FEDER) UE, under contract TIN2017-84875-P and partially funded by the Fundacion Escuelas Universitarias Gimbernat (EUG).

We thank Román Bond, research engineering of Universidad Nacional Arturo Jauretche (Argentina), for his support in the implementation of the simulator.

REFERENCES

[1] D. Encinas, M. Naiouf, A. De Giusti, S. Mendez, D. Rexachs, and E. Luque, “On the Calibration, Verification and Validation of an Agent-Based Model of the HPC Input/Output System,” in SIMUL 2019, The Eleventh International Conference on Advances in System Simulation, 2019, pp. 14–21.

[2] D. Encinas, M. Naiouf, A. De Giusti, S. Mendez, D. Rexachs and E. Luque, “Análisis funcional de la pila de software de E/S paralela utilizando IaaS,” VI Jornadas de Cloud Computing & Big Data (JCC&BD), 2018, pp. 1–6. [Online]. Available: http://sedici.unlp.edu.ar/handle/10915/69465. Retrieved: 08/2020

[3] Sharcnet. Parallel I/O introductory tutorial. [Online]. Available: https://www.sharcnet.ca/help/index.php/Parallel_IO_introductory_tutorial. Retrieved: 10/2019. (2017)

[4] R. Thakur, W. Gropp, and E. Lusk, “Data Sieving and Collective I/O in ROMIO,” in Proceedings of the 7th Symposium on the Frontiers of Massively Parallel Computation, ser. FRONTIERS ’99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 182–189. [Online]. Available: http://dl.acm.org/citation.cfm?id=796733. Retrieved: 10/2019

[5] W. Loewe, T. McLarty, and C. Morrone. IOR Benchmark. [Online]. Available: http://sourceforge.net/projects/ior-sio. Retrieved: 11/2020. (2015)

[6] A. Núñez, J. Fernández, J. D. Garcia, F. Garcia, and J. Carretero, “New techniques for simulating high performance mpi applications on large storage net,” J. Supercomput., vol. 51, no. 1, Jan. 2010, pp. 40–57.

[7] J. Kunkel, “Using Simulation to Validate Performance of MPI(-IO) Implementations,” in Supercomputing, ser. Lecture Notes in Computer Science, J. M. Kunkel, T. Ludwig, and H. W. Meuer, Eds., no. 7905. Berlin, Heidelberg: Springer, 06 2013, pp. 181–195.

[8] N. Liu et al., “Modeling a leadership-scale storage system.” in PPAM (1), ser. Lecture Notes in Computer Science, R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Wasniewski, Eds., vol. 7203. Springer, 2011, pp. 10–19.

[9] B. Feng, N. Liu, S. He, and X.-H. Sun, “HPIS3: Towards a High-performance Simulator for Hybrid Parallel I/O and Storage Systems,” in Proceedings of the 9th Parallel Data Storage Workshop, ser. PDSW ’14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 37–42.

[10] Lustre Simulator. [Online]. Available: https://github.com/yingjinqian/Lustre-Simulator/tree/master/doc. Retrieved: 06/2020. (2016)

[11] C. Carothers, D. Bauer, and S. Pearce, “ROSS: a high-performance, low memory, modular time warp system,” in Fourteenth Workshop on Parallel and Distributed Simulation., 2000, pp. 53–60.

[12] V. J. Julián and V. J. Botti, “Estudio de metodos de desarrollo de sistemas multiagente,” Inteligencia Artificial. Revista Iberoamericana de Inteligencia Artificial, vol. 7, 2003, pp. 65–80. [Online]. Available: http://www.redalyc.org/articulo.oa?id=92501806. Retrieved: 10/2019

[13] A. Borshchev and A. Filippov, “From system dynamics and discrete event to practical agent based modeling: reasons, techniques, tools,” The 22nd International Conference of the System Dynamics Society, Oxford, England, 07 2004.

[14] E. Kremers, “Modelling and Simulation of Electrical Energy Systems through a Complex Systems Approach using Agent-Based Models,” Ph.D. dissertation, Universidad del País Vasco (UPV/EHU), 2012.

[15] M. Taboada, E. Cabrera, F. Epelde, and E. Luque, “Using an agent-based simulation for predicting the effects of patients derivation policies in emergency departments,” in International Conference on Computational Science, Barcelona, Spain, 2013, pp. 641–650.

[16] A. Uselton, “Deploying Server-side File System Monitoring at NERSC,” in Proceedings of the 2009 Cray User Group, 2009.

[17] Lustre Manual. [Online]. Available: http://doc.lustre.org/lustre_manual.xhtml#idm140436306123424. Retrieved: 06/2020. (2017)

[18] Admon Guide. [Online]. Available: https://www.beegfs.io/wiki/AdmonGuide. Retrieved: 06/2020. (2019)

[19] Fresh Open Source Software Archive. [Online]. Available: https://fossies.org/linux/orangefs/doc/pvfs2-logging.txt. Retrieved: 06/2020. (2017)

[20] A. M. Law, Simulation Modeling & Analysis, 5th ed. New York, NY, USA: McGraw-Hill, 2015.

[21] PVFS2 Team, “PVFS 2 File System Semantics Document,” PVFS Development Team, Tech. Rep., 2015.