Master in High Performance Computing

Discrete Simulation of DDN IME® for architecture prototyping

Supervisor(s):
Jean-Thomas Acquaviva,
Stefano Cozzini

Candidate:
Daniele Tolomelli

3rd edition
2016–2017

Contents

Chapter 1

Introduction

1.1 Problem Explanation

1.1.1 Device presentation

Computer memories are designed to fit particular needs, based on their size or latency. The memory hierarchy makes every program able to write the most accessed data from a lower latency memory in order to access more often the smaller and faster memory. Every time some data is not present in a lower level memory, the system must read the higher latency one, wasting CPU cycles waiting for the memory to complete the operation. To avoid or to mitigate this issue an additional layer can be added. This is what happened with L3 Cache: it is not needed for the computer to work, but to make it faster, caching more data at the same time.
With the same idea Data Direct Networks developed a system that acts as a cache between main memory and mass storage, filling the gap that there is right now. This system is called IME, the Infinite Memory Engine.
As a cache it operates in a transparent way, making every already developed program able to run on on this platform.
The device can intercepts IO calls in many ways so that its users can choose the most appropriate method for their programs.

1.1.2 The problem

DDN IME as of now has some scaling issues when a large amount of servers are installed. To keep data consistency, the system has to perform some transactions from server to server. This traffic will decrease the available bandwidth for the clients limiting the total amount of servers, hence the maximum available cache memory.

Moreover inside the system are present a lot of free parameters, which will be discussed later, that can affect performances. At the current time we are unable to determine their optimal value. Multiple runs with different configurations will help the investigation to the optimal configuration based on a given architecture.

1.1.3 The solution

The solution proposed by DDN is a simulator of the system, enabling the fast exploration of different solutions without the need to actually build many systems.
Different kind of simulators exist. Below will be explained some of them:

Simulator	**Details**
Analytical	Every event is weighted through a function and a final formula should lead to the result. The precision is based on the error of each function from the real value
Trace-based	The simulation is based on a trace generated by the environment we want to simulate. The input is the result of the operations. Can take much storage space
Execution Driven	Instead of taking result from a trace, the simulator actually perform some task. Instead of taking storage space takes time because actions are actually performed
Event based	Instead of simulating time passing the events are scheduled and called consecutively. Suitable if events do not happen regularly or at a regular interval
Discrete Event	Time evolve by 'hop'. The clock of the system is increased when an event occurs. The completion of an event is defined by a transition function. The flow of event is defined by a finite state diagram. Time is split in small units. As an event happens, there is a change of state in the system following a finite state machine diagram.
Continuous Event	Time is continuous in contrast with discrete event simulator.

From these simulator, the most suitable for the case is the **Discrete Event Simulator**. The analytical simulator is suitable for an environment where some events can be parametrized, but IME is a complex environment

with resources and network fabric that can be busy when required.
For trace-based and execution driven are required both data and machine configuration and none of them are available since IME is a new hardware.
Event based simulation can represent better the environment, more precisely a discrete event simulator is more effective since is not needed a continuous time space, avoiding rounding errors due to binary representation.

1.2 Requirements Analysis

Simulators are not necessarily computationally intensive applications. The computational complexity depend on the accuracy of the environment to be simulated. In this case the simulation of network transactions will end up in waiting an amount of simulation time based on a formula. Such approach allows to focus on flexibility, where the system can easily evolve on by a component basis. For instance a more accurate model can be implemented if it appear that the modeling of the communication layer is injecting too much noise. Following the principle of "Not reinventing the wheel" the project has started using a library for discrete simulation. This accelerated the project since at this point only the logic of the system has to be implemented. After the development part will come the investigation phase to better understand every free parameter inside the system. To list the requirements, these are:

- environment focused on flexibility
- compatible simulation library
- monitor tools to inspect the system performances
- test-suite for the simulated system

Every good practice about software design and development is implicit in this list.

1.2.1 HPC Relevance

The simulator by itself is not the product relevant from the HPC point of view. What is relevant instead is the problem it is going to solve: there is already a problem of scalability of a product and the simulator should solve or help to solve it faster than a traditional development process would. This will accelerate architecture and configuration investigation.

1.3 Development environment

To satisfy the requirements we chose Python because of its flexibility and an already implemented library. This platform is well suited to create many models, focusing the attention on the system architecture instead of low level details.
The simulation library chosen is SimPy [7], a discrete events simulation library. It is well maintained and perfectly integrated with language.
A drawback of a Python environment is its dynamic types: this is not a problem in general but one of the main features of python instead that makes it one of the best scripting language. Anyway for bigger projects a safe type checking helps the development. The lack of this checking by design slowed down the development at the beginning.

1.4 SimPy

This discrete event simulation library is *Events* oriented. An event can be processed and can be represented as a function performing some general action, based on the situation. The caller event can wait for the completion of the task called or just trigger it and simulate an async call. This make a simulation of a multithreaded environment really simple using a few lines of code.
More details on this library will be explained later, in section 3.2

Chapter 2

IME Architecture

IME system is based on a Client-Server interaction: every compute node has a Client side process intercepting IO communications that are sent to IME servers.

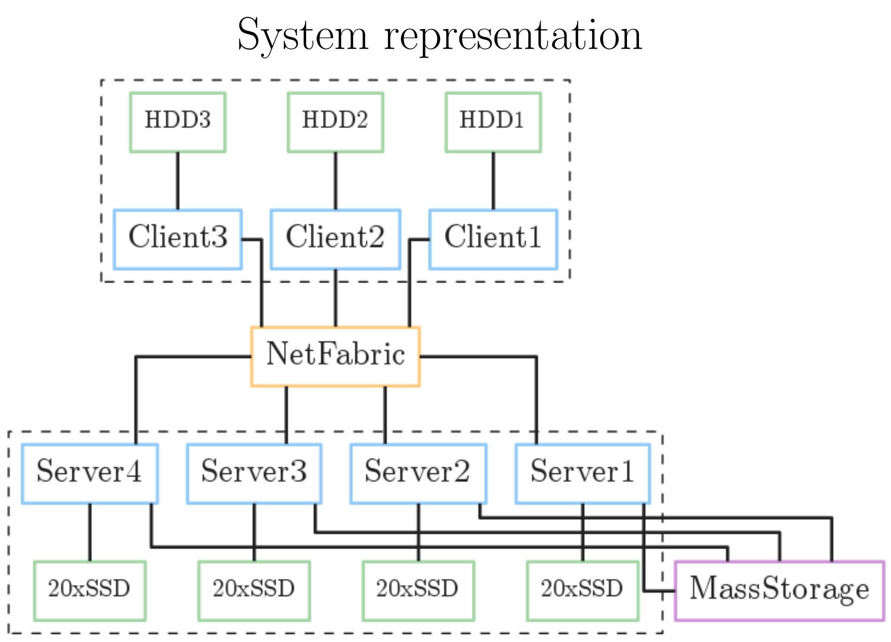

Figure 2.1 Graphical layout of an IME node. Clients have a local disk and a connection to the IME servers through a Network Fabric. Each server has a set of SSDs installed acting as a cache

In a possible configuration , as shown in figure 2.1, one can identify the different components:

- 3 clients with a single HDD each. This hard drive is used to store the local image of the OS. In modern data center, client nodes tend to be disk-less (they boot over the network). Local HDD can be used to store the /tmp and the /var and various Linux system log mechanism. But applications are doing most (if not all) of their I/O using the fabrics.
- a Network Fabric that interconnects every component of IME. Specifically in IME is installed an EDR Infiniband architecture.
- up to 20 SSDs per server, that can store data faster compared to the single client's HDD

Given the skeleton of the architecture, can be added or removed clients, servers and disks for each one. This changes will end up in different performances that the simulator must be able to detect.
The aim of this simulator is to detect the bottlenecks in order to gain knowledge on which parts need to be improved allowing the system to scale. Every configuration and test will be discussed in depth in the analysis section (5).

2.1 Network communication time

Network communication is one of the main tasks that are committed in this system, so an accurate model to represent it is needed.
The parameters that determine the time elapsed in a network transaction are *latency* and *bandwidth*. A file theoretically is sent dividing its size by the bandwidth available. In the real world we have to consider also the latency present inside the system: the smallest file possible will be sent only after $< latency >$ amount of time.

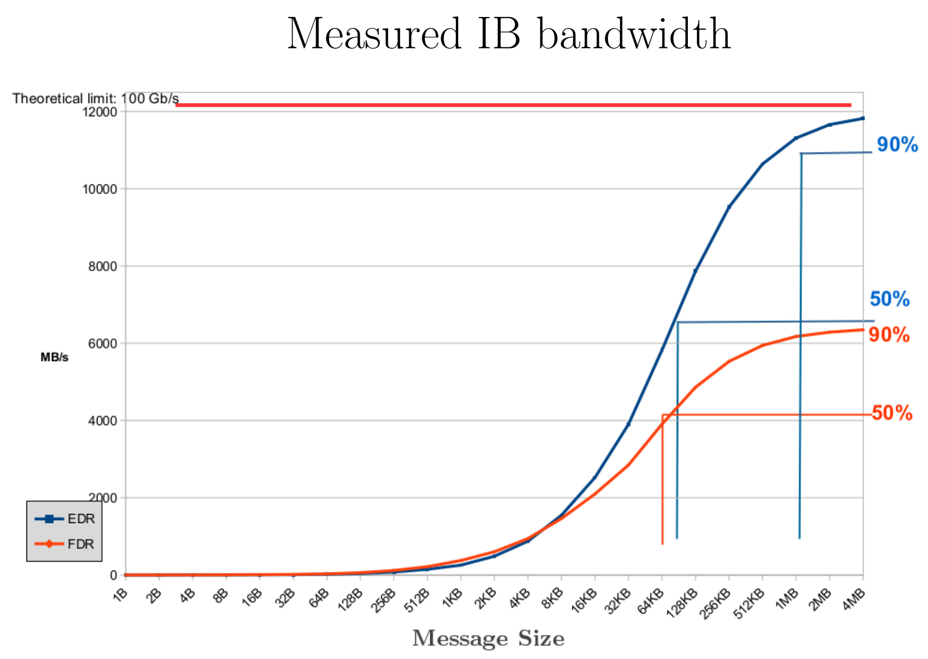

Figure 2.2 Infiniband network performs differently based on the requests size involved in the transaction. The plot represents different performance based on IB architecture and request size. IME has installed EDR Infiniband so the upper blue curve represents the actual IME network speed [4]

From figure 2.2 we know that the optimal packet size for communication over EDR infiniband is 1MB.
IME Client packs together requests smaller than this threshold and split bigger files to achieve this communication pattern. This is further referred as **Network Buffer**. If requested explicitly, every request can be flushed in order to complete a communication in case of remainders. This sets a domain over the size of the packet sent: we will always have packets that are in the interval $]0, 1024]KB$. The model should predict correctly the time required by any network communication inside this interval.

file size (KB)	time required
0	*latency*
1024	*1024KB / bandwidth + K * latency*

These are respectively the worst and the best usage of the network.

The case with an empty file is only ideal to show the role of the latency in the equation, removing any data transaction. To replicate the behaviour of a real network infrastructure we need a function with a diagonal limit, namely $\lim_{x\to+\infty} y(x) = +\infty$, that is a real behaviour for every network.
The function of choice then is an *hyperbole* adjusted to match the specified coordinates.

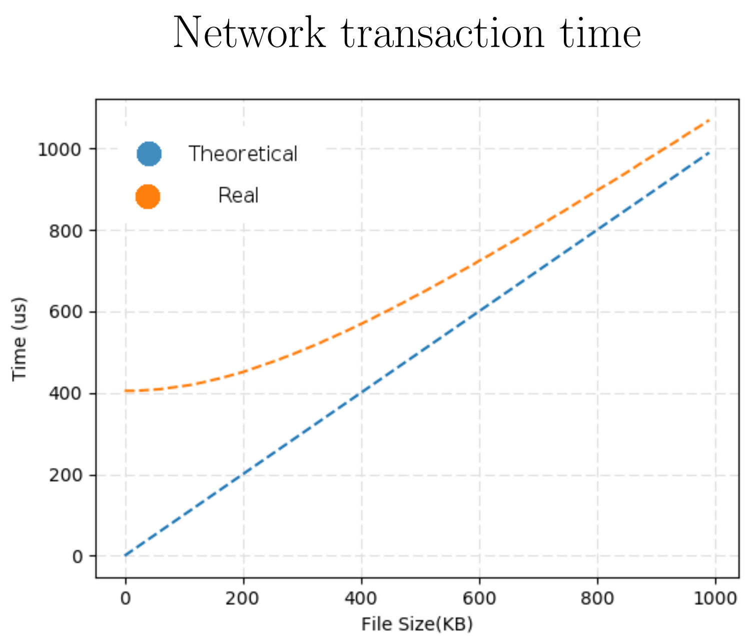

Figure 2.3 The diagonal limit function applied in the network and disk performance model

2.2 Tokenized communication

IME has been designed to be employed with a large number of both clients and servers. This fact leads to many parallelization problems and resource management. With a large amount of clients the network bandwidth will run out very quickly. To avoid denial of service or resource starvation, IME is using tokenized communication, the standard way to implement throttling. The concept of the token is similar to the Token Ring communication [3] that prevents a single client to use all the resources of the system and sharing equally the bandwidth among all the clients. Here instead of having a single token shared over all the clients, each client has a fixed amount of tokens. The token usage is the following:

- Consumed when making a request
- Recovered when receiving an answer

This means that for n token, a single client can have n communications in parallel. So far DDN already set a value of 24 tokens that are not completely used in real applications, meaning that there is not a bottleneck, but is anyway a free parameter that can be inspected and adapted in the simulator.

2.3 File queuing

Every clients has internally a write queue for every single server. As it is asked to write a file, the client will scatter the single file to multiple queues in order to make use of most servers and speed up the transaction. When a client wants to perform a write it has to care about:

- The amount of servers that will be involved depending on the file size
- Not scattering too much a single file to avoid unnecessary broadcast operations. When the client later will need to read the file, it will have to generate a large amount of requests to read a small piece of data.
- Perform small transactions of 1MB to maximize network bandwidth

Since this system involves not only disk operations but also network ones, we must be careful to not deplete the network resources. This means that to maximize an IO operations, make sense using as many devices as possible, but in a network environment is better to reduce at minimum the packets emitted.
To overcome these opposing behaviour, IME has 2 layers of file queuing: **bucket**-based and **buffer**-based queuing systems. A bucket is considered as an interval of data of a fixed size. A single file is split in bucket-sized parts and then queued to be split again to buffer-sized parts, in order to perform a network communication.

Queueing system using buckets

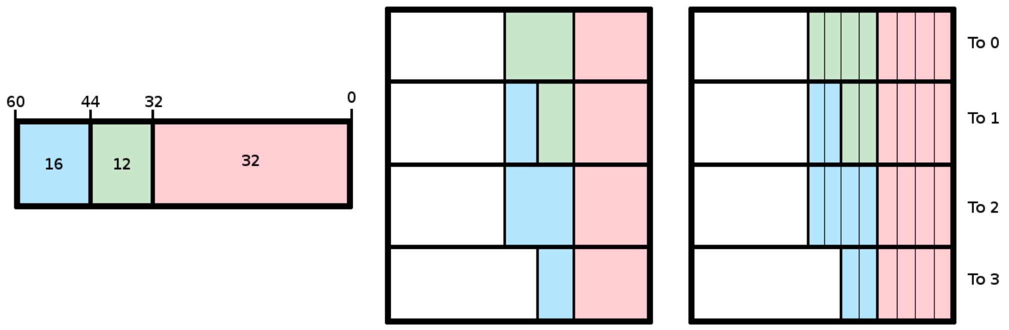

Figure 2.4 Files to be sent are initially queued. To decide their targets, or their parts if splitting is needed, these are queued considering chunks bigger than network buffers, called *buckets*. Then every bucket is split in network Buffer sized chunks and later sent

- 4 servers are installed
- the user asked to write every file, 32, 12, 16 MB respectively, in chronological order.
- Bucket size = 8MB
- Buffer size = 2MB, just for plotting purposes
- parity is not considered

In figure 2.4 the first queue is the file-oriented one: every file as it seen by the client point of view. Then the central box represents the queue to each server with the bucket view of the files. The last one represents also the server's queues but with network buffers instead of buckets.
This different granularity allows to make use of the server's devices if a file is big enough, saving at the same time network operations when the user asks to read those later.

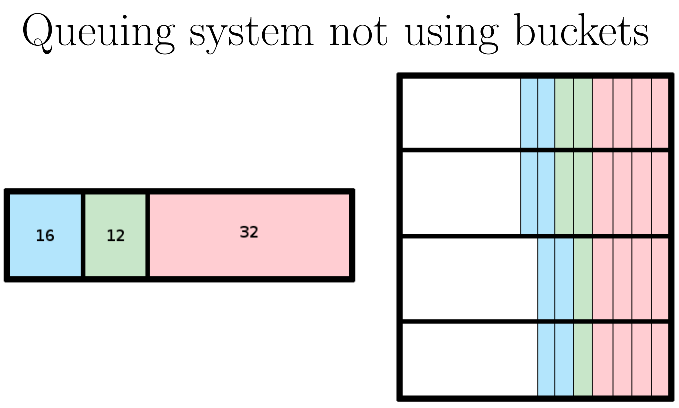

Figure 2.5 If buckets are not used, a file would be more spread over the servers, meaning more read requests generated and a higher network usage.

The upper image shows how packets should be distributed without the bucket queuing layer. Note that, from the buffer point of view, the green file could be split over all the servers, that translates in great write operation but poor read operation, since it has to perform a broadcast over all the system to make a request of it. This example must be thought with bigger numbers: in a case with 50 servers, with a buffer point of view, a file of 50MB involves a broadcast, while the bucket point of view mitigate this behaviour increasing the file size required by a broadcast to 400 MB.
Some examples in the following table show the *number of read requests generated* from a single client reading a file, using the bucket view or not where 50 servers are installed. The two different results are based on data alignment, whether my request will be satisfied by an entire buffer/bucket or two parts of them. For example a request from 2 to 10 is 8MB but will be satisfied using 2 buckets.

	Number of requests using	
Data read	also buckets	only buffers
1 MB	1,2	1,2
8 MB	1,2	8,9
48 MB	6,7	48,49
400 MB	50	50

For requests size below 400MB there are benefits reducing the number of read requests generated and so the network traffic. For requests bigger than 400MB we cannot see the difference since network traffic reduction has been overcome by IO operations.

2.4 Distributed Hash Table (DHT)

IME servers must be aware of the data that are inside the system, being also fault tolerant in case of failures. These should also be careful of not flooding the network with too many requests.
DDN used a DHT to solve this problem, allowing the scaling of the number of servers and allowing the failure of a single one.
As a file is stored in a server, its metadata, the way to access it, are sent to another server responsible of delivering the accidentally lost data.

2.5 Server side write

As the server receive a buffer, a number of steps are performed before sending back an ACK to the client.
Two separate task are performed, divided in two different planes: **Control plane** and **Data plane**

2.5.1 Control plane

Here is performed metadata propagation if necessary and done some *logging* operation to a local journal device.
Metadata propagation involve the communication of the current metadata to another machine, meaning that this operation has to wait an ACK before being completed. This can hit very hard the performances, forcing an additional network communication for every write. So this operation is cached instead and done as one of the following condition is met:

- Client requested an eager commit
- Cache limit reached. After 128 requests is sent a single request instead of 128
- Timeout triggered. If no write happened recently, current data must anyway be saved so is forced a propagation

Metadata propagation is a necessary step in order to recover data in case of failures: the metadata can be read from the DHT of another working server, then data can be recovered asking to every involved server the lost parity groups.

2.5.2 Data plane

Data received is stored locally as well as metadata. Since data and metadata are accessed in a different pattern, and metadata represent a bottleneck when it comes to files reading, metadata are stored in dedicated devices, less capable but with wider bandwidth.
The problem to face in this case is to make use of every device understanding the way every device works.
As for network communications, the diagonal limit behaviour (see section 2.1) can be applied also in this case. Then we look for bigger transactions, avoiding small ones.
The request is seen in a twofold way:

- As a set of file parts. These informations are stored as metadata, in order to know exactly where each file part is stored. In the picture this is represented by the upper layer. In this case the user sent files *a,b,c,d,e* and *f*

- As a byte stream: aware of the behaviour of HDD we are now interested in the data as a set of bytes to be written to the disk, nothing more. Metadata will tell us the exact location of a file, but we know want to perform the most efficient operation.
  In this step the network is split in **CML_oid**s, chunk of 128KB. Each of these will be stored to different device, if possible, in a round-robin fashion to distribute equally the load.
  In the picture the CML_oid-s generated are the light blue boxes

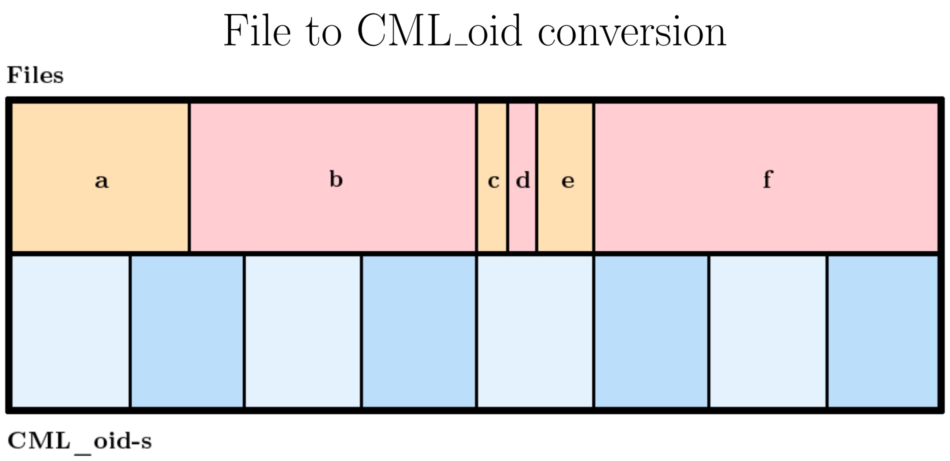

Figure 2.6 The received files are stored as CML_oid, a byte-level view of the incoming data. Every CML_oid can store a fixed amount of data

2.6 Data Read

If a client wants to read a file, or a part of it, it interrogates its file map to make a request to the interested server, saving network communication avoiding a broadcast if not necessary.
After the server received the request, it has to interrogate its local DHT in order to know from which devices has the requested data. A read operation is then performed on those devices and data is sent back to client using as always a buffered communication of 1 MB per packet.
This is the optimal work flow, that is without any kind of error.

Chapter 3

Development

Since IME is a complicated environment, has been chosen to develop the simulator using an iterative approach. This allowed an understanding of the whole system showing a single part at a time, at the cost of creating prototypes that needed to be rewritten partially or completely.
To speed up this prototyping approach has been used the Dot language [1] to quickly develop finite state machines that will have been later implemented.

3.1 Environment choice

Based on the conditions explained in the introduction (see section 1.2), *Python* seemed a reasonable choice. Mostly because of its flexibility to changes, not worrying about small detail like memory addresses and garbage collection. Since this application is not computationally intensive, not involving any kind of floating point operation, the cost of an interpreted language is not so high.
Moreover modern features coming from Python 3.5+ made easier to develop a relatively big applications, as this simulator is, using types annotations and better generators support.

3.2 Simpy

Simpy is the library the projects is based on. It is a Discrete Simulation library that fulfills the requirements. Since it exploits language mechanics like generators does not add overhead on the application. It is process-based meaning that every task can be scheduled as a process, not waiting for others task to be completed. The library is anyway serial, not creating threads by itself. Anyway it can be integrated using other libraries, like *threading* (see

[2]), but so far the simulation achieves reasonable wall times, so there is no need to parallelize it.
Simpy offers a framework in which our routines are executed and tracked. The status of an agent inside the system is tracked using python generators, which are lazy iterators that generate the requested iterator only when needed. Because of the framework environment we need specific functions that return a generator that Simpy can take care of and others functions that keep track of the system status. Simpy can handle a new agent if a function returns a new generator. Based on this new generator, a new agent is created, able to make the virtual clock going on, to create other virtual processes or to wait for other processes to be completed, according to the situation. Another set of functions are required to keep track of the status of the system.

3.2.1 SimPy Components

The philosophy of Simpy led to the development of ready-to-use components that ease the process management like mutexes and task queues. Anyway Simpy provides only tools to manage its processes, lacking any structures to collect usage data for investigation at the end of the simulation. For this reason additional tools has been developed both in Python, for internal data collection, and Bash to automate plotting and file management.

Timeout

The library has some predefined events like a *Timeout* event that just simulates time flow for the caller.
This call blocks only the calling agent, allowing every other event already triggered to be processed correctly.
Every real time consuming action is simulated through this function: the call with the correct amount of time will make the calling agent to wait until the task is virtually completed.

Resources

In every simulation environment there are components that need to be accessed by many agents and to process correctly this resource contention. This means keeping a queue of every agent and triggering one as a resource is freed.
SimPy takes care of this queue management, providing many types of resources, fitting many needs.

- **Resource** [8]: allows multiple agents to access this tool at the same time. A request to this resource is blocked if the quantity of agents using it is already at the maximum value. The maximum number of agents can be set in its constructor. Setting `capacity = 1` means this resource will behave as a mutex.
  An example is accessing a disk to write a file: only a single file can be written at once, so the device's resource will have a capacity set to 1. The writing process must obtain access to this resource before proceeding further.

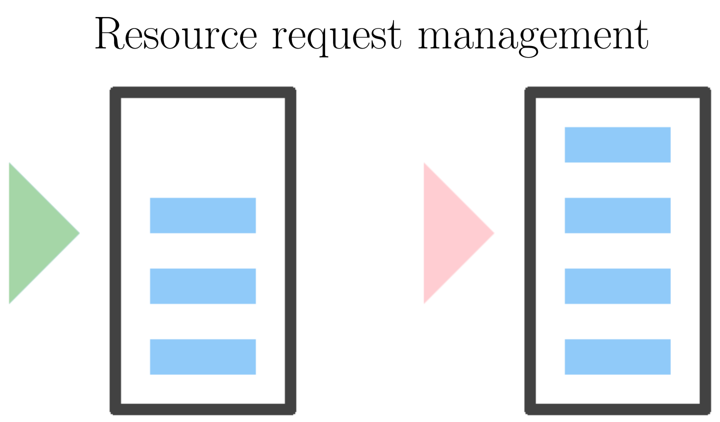

Figure 3.1 Request management using simpy.Container. A *Resource* with still space accepts a new request (left). A full one rejects the request instead (right)

As shown in Figure 3.1 the requests are accepted based on the current load of the resource. If there is still room for another request, as happens in the left case, the request is accepted.
Otherwise if the request is completely filled, the request is blocked until some of the already occupying request free the resource.

- **Container** [6]: whereas the "Resource" tracks the number of accesses to a resource, the container is concerned about the quantity of a resource stored. As a container cannot store bigger amount of goods than its size and cannot give more than it has stored already.
  An example is shown if Figure 3.2 where different put and get requests are performed. Put requests (on the right) are allowed if there is enough space, get requests are allowed if there is enough good stored.

As a side note, the library still lack custom callback implementation to handle blocked requests. This means that there is not the possibility to perform some actions if a container is too full, blocking some requests. This forces to not fully rely on SimPy classes and implement custom solutions that benefits from these callbacks.

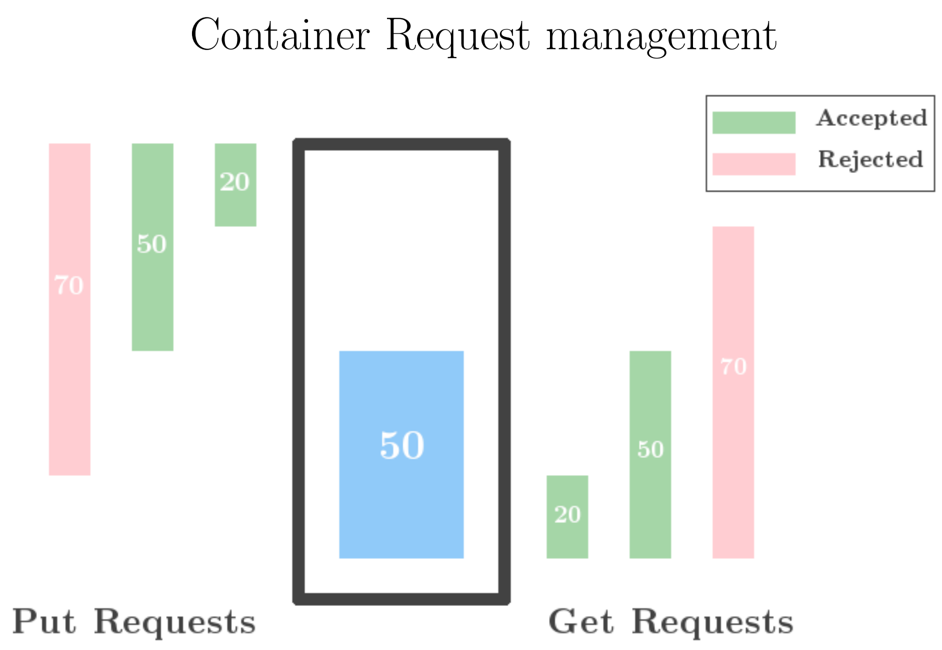

Figure 3.2 A simpy container with a capacity of 100. On the left the requests to put a good in the *Container*, allowed only if there is enough space. On the right the get requests, allowed only if there is enough resource currently stored in the container.

3.2.2 Processes Management

Inside the SimPy framework in order to simulate a behaviour a task is seen as a *process*. This is just a concept: a SimPy simulation runs on a single system process. After having instanced one or more processes, the caller is able to manage its flow based on the nature of the situation. It can:

- instance one or more processes an proceed with the flow, in case the call should not wait for the completion and the task created.

- wait for the completion of the single process, in case the caller should wait for the completion of the task created
- create more processes. Proceed with the flow as soon as *only one* finishes its execution
- create more processes. Proceed with the flow when *every one* finishes its execution

3.3 Tools created

Since SimPy provide only simulation utilities, lacking data gathering components and plotting, these features has been developed aside. Specifically has been designed:

- **Logger**: in order to gather data about the components, every action is registered using a shared object. Every step is registered, in terms of time spent on that action or in times an action is performed. At the end of the simulation the data gathered are printed to external files, allowing parsing and plotting using external resources.
- **Testing environment**: to inspect the behaviour of the system should be easy to specify many test cases using different configurations. For this reason bash scripts and makefiles has been developed to manage the simulator at a higher level. At the end of the multiple simulations, plots are generated based on the data gathered from each run.

3.4 Profiling

Time spending operations are simulates as already specified. The simulator is not a computationally intensive application but as things start to grow, the wall time increases. This because of the large amount of data needed to run the application: a file of 1 GB must be split in buckets, network buffer and finally in CML_oid of size 128 KB. This result in $1GB/128KB = 8192 CML\_oid$ allocated just for a single gigabyte of data written. Some experiments on C integration has been conducted but the real problem is data allocation and its management. The usage of suitable data structures for every need plays a key role in speeding up the simulator.

3.5 Alternative Python Implementations

Looking for performances is reasonable to try different implementation of the python interpreter. The official one is CPython, but other implementation for different usage are available. Some of them are

Name	**Description**
Jython	Implementation that runs on the JVM. Python code is compiled into bytecode enabling better performances. Also removes the Global Interpreter Lock making use of real threads.
IronPython	Just as a matter of compatibility, a version of Python that integrates with Microsoft .NET technologies
Pypy	Distribution with a built-in JIT compiler, overcoming the purely interpreted nature of python

Many other implementations exists, but more for integration into specific environment, like IronPython does, than achieving a different behaviour of the interpreter.
The choice of Jython has been dropped since it's compatible only with python 2.7. From the beginning the simulator has been developed in Python 3.5 due to useful features for the development of a big project and some utilities on the generators.

3.5.1 Pypy

Trying a different implementation, Pypy has been chosen. It provides a custom version of numpy, but the simulator does not make use of numpy. Many tests showed that an appropriate container from the package `collection` behave better that numpy arrays. The custom numpy installation is the only step required for Pypy to work.
The features advertised by Pypy are:

- *Speed* using its JIT compiler. Anyway every time the application is called, the compilation phase must be run again.
  On their website Pypy team advertise a speed up over 7x compared to CPython 2.7. Tests on this specific simulator will follow.
- *Less memory usage* using a better custom garbage collector
- *Compatibility* so it can run any existing python applications

Testing the actual performances of Pypy over CPython I run the simulator with a reasonable large request pattern: a single client asked to write

8192 files of 16MB. Since Pypy should use also less memory, main memory contention between different processes should be mitigated. So multiple requests of the same size have been launched in parallel using `xargs` command. The simulator is serial, but more requests can be processes at the same time. For this reason the usage of `xargs` is required.
The tests has been done using an Intel i5 4690 3.5GHz 4core/4threads with the graphical session turned off to avoid data fluctuation due to other processes in the system.

Time (s) using CPython 3.5 Processes in parallel	First	Second	Third	Fourth	Total
1	41.91	41.97	42.10	42.00	167.98
2	43.61	43.32	43.40	43.70	87.02
4	52.56	54.32	54.27	54.05	54.32

Time (s) using Pypy 3.5 Processes in parallel	First	Second	Third	Fourth	Total
1	69.29	69.31	69.31	69.65	277.56
2	71.92	72.77	72.41	72.37	144.33
4	92.03	92.06	92.60	92.05	92.60

What is clear from the data gathered is that Pypy is not working for our case. Further discussion will be done in section 3.6.
Using a parallelization of 1, the best times are achieved since there is no main memory contention. As the number of processes start to grow, more contention happens but since these times are in parallel, they take less time overall.
Measuring this phenomenon a result can be extracted from the data using the formula $(max(P_4)/min(P_1)-1)*100$ where $max(P_4)$ is the worst process with a parallelization of 4 and $min(P_1)$ is the best process with no parallelization. For the *total* case every process can be expressed using an average value of the times collected so $total\_time_4 * 4/total\_time_1$ The results follow

Memory contention degradation	Single	Total
CPython	29.6%	29.3%
Pypy	33.6 %	33.4%

This data shows how Pypy behaves better increasing the parallelization. The different is not critical, but since Pypy is an alternative implementation, for this specific application there is no reason to choose Pypy over the default CPython.

3.6 Flamegraph

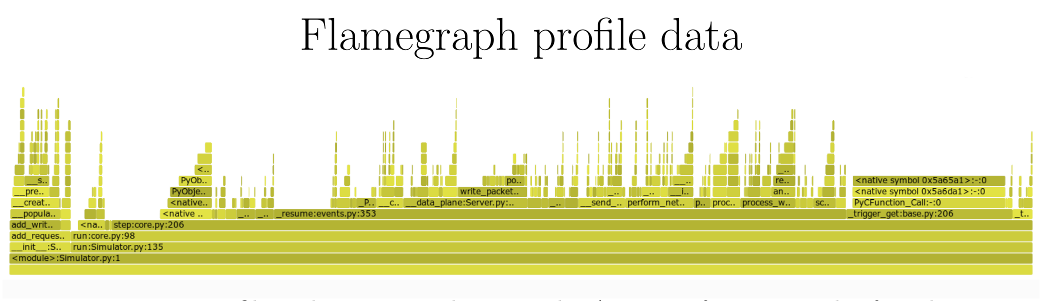

Figure 3.3 Profiling data using Flamegraph. A vector format can be found at https://drive.google.com/open?id=1JOMhRwSq58nZgOFQMo3JHjHOb6mjI7jh

The profiling task has been accomplished using Flamegraph [5] a stack trace analyzer generating an interactive vector graphic to inspect graphically the application.
Figure 3.3 represents the flamegraph of the simulator performing a write of the file requested, a read operation of every file written and a recovery operation due to a disk failure.
To better inspect the result, a text version of the data plotted has been provided. The following are the 20 most expensive functions that are processed inside the simulator.

Time(s)	**Source File**	**Function**
6310	core.py	run
6016	core.py	step
4273	events.py	_resume
1007	Server.py	__data_plane
695	events.py	__init__
635	core.py	schedule
593	ServerManager.py	perform_network_transaction
560	base.py	_trigger_get
557	ServerManager.py	add_requests_to_clients
533	DataIndexer.py	write_packet
525	Server.py	process_write_request
454	Client.py	add_write_request
445	Client.py	__populate_bucket_queue
428	Client.py	__create_buffers_from_buckets
415	events.py	__init__
410	Client.py	__send_buffers
403	base.py	__init__
373	Client.py	__send_write_request
357	resource.py	__init__
357	Client.py	__prepare_write_request

Files with a capital letter are from the simulator, lowercase ones are from SimPy. These are not all different functions: there is the stack to be considered, so for example the *run* function includes most of the functions above. Anyway a non negligible part of the simulation is composed by overhead introduced by the framework, caused by the resource and processes management, aside from their execution.
Must be noted that multiple solutions can be proposed leading to the final results. Some of these solutions in early prototypes were creating many SimPy processes in order to simulate the environment. Later this kind of approach has been dismissed since it was introducing a noticeable overhead inside the framework. Has been adopted an approach with a single agent that can solve more tasks on his own, using a single SimPy process.
Moreover the time of many tasks could be computed ahead of time avoiding object instantiation, but doing so takes the simulator closer to an *Analytical Simulator* behaviour, that is not the environment of choice. The only components that can be simulated in such a way are those that do not need to wait for shared resources and can be processed on their own.
So the overhead that SimPy is causing right now has already been reduced to its minimum.

Chapter 4

Advanced IME Topics

4.1 Erasure coding for data loss

In CPU caches, whenever happens a cache miss, data is read from main memory instead. No data can be potentially lost. In the worst case can happen a cache miss forcing the lower layer memory to transfer data from the upper layer memory.
IME acts as a cache too but for the problem it is solving, there is no communication between the source and the target. This means that if data is lost inside IME, it is *permanently* lost. IME has a system to overcome this problem that is *Erasure Coding*.
As it happens in RAID level 4, 5 and 6, what is stored in the disks is not just data but also an added amount of data based on the original data. In case of loss of one piece of data, this missing part can be reconstructed comparing remaining data and parity.

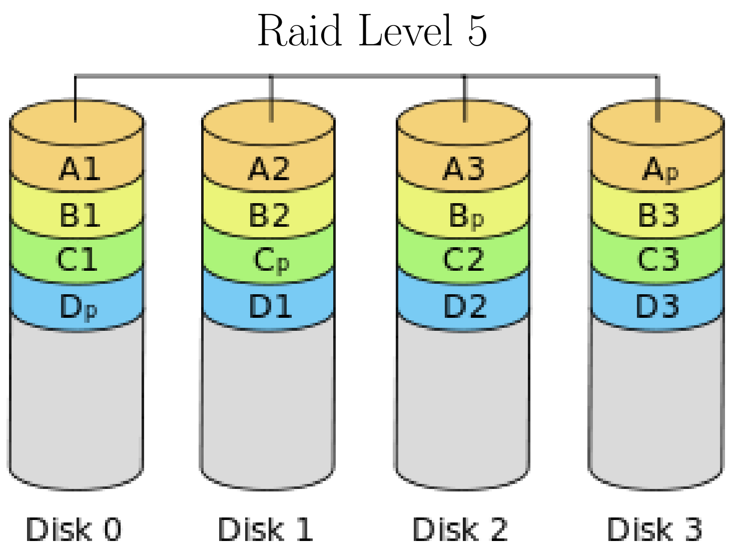

Figure 4.1 In RAID 5 data is interleaved with parity bits. If a single disk is corrupted, missing data can be recovered comparing remaining data using parity

For this correction system to work, we need to make associations between groups of data that must be grouped together later when data loss happens. IME considers a single block of data a **Network Buffer** (see section 2.1) and the group that will contribute to data recovery as a **Parity Group**.
As an example, from figure 4.1 A1, A2, A3 and A_p are individually a Network Buffer. All of them grouped together makes a Parity Group.
Parity generation is a process that happens on the client that want to send the packets, since it has to create requests that must be aware of the location of the packets belonging to the same parity group.
Is essential that every piece of the same parity group are sent to different server: the loss of a single server could otherwise means the loss of multiple piece of data. If every part is stored in a different server, the inactivity of a single machine is not a fatal error for the system.
Parity be generated using different methods, meaning that the system can sustain the loss of multiple piece of data keeping the ability to recover the lost data. This is an option that can be set or not based on the needs of the user.

4.1.1 Handling remainders

Using this system to recover data, a bond is created among parts of the same parity group: not only these parts share the same parity group-id but also must be sent at the same time to the servers. If the packets of the same parity group are not completely delivered to the servers, these cannot recover this group in case of data loss.
Because of that the client must wait until it has enough data to send, filling a set of network buffers. If it does not have enough data, it waits until more write requests are needed. The send operation can anyway be forced using an **eager commit**, sending buffers not completely filled or even not enough buffers to fill a parity group.
Parity data is then generated using the data available.

4.1.2 Parity generation

Parity data is generated for every parity group created based on the **geometry** set. Geometry specify how many, if any, parity packets are generated per group. Geometry is specified as $D + P$ whereas D represents data packets per group and P represents parity packets per group. So a geometry of $3+1$ specify that a group is composed by 3 data packets and 1 parity packet.

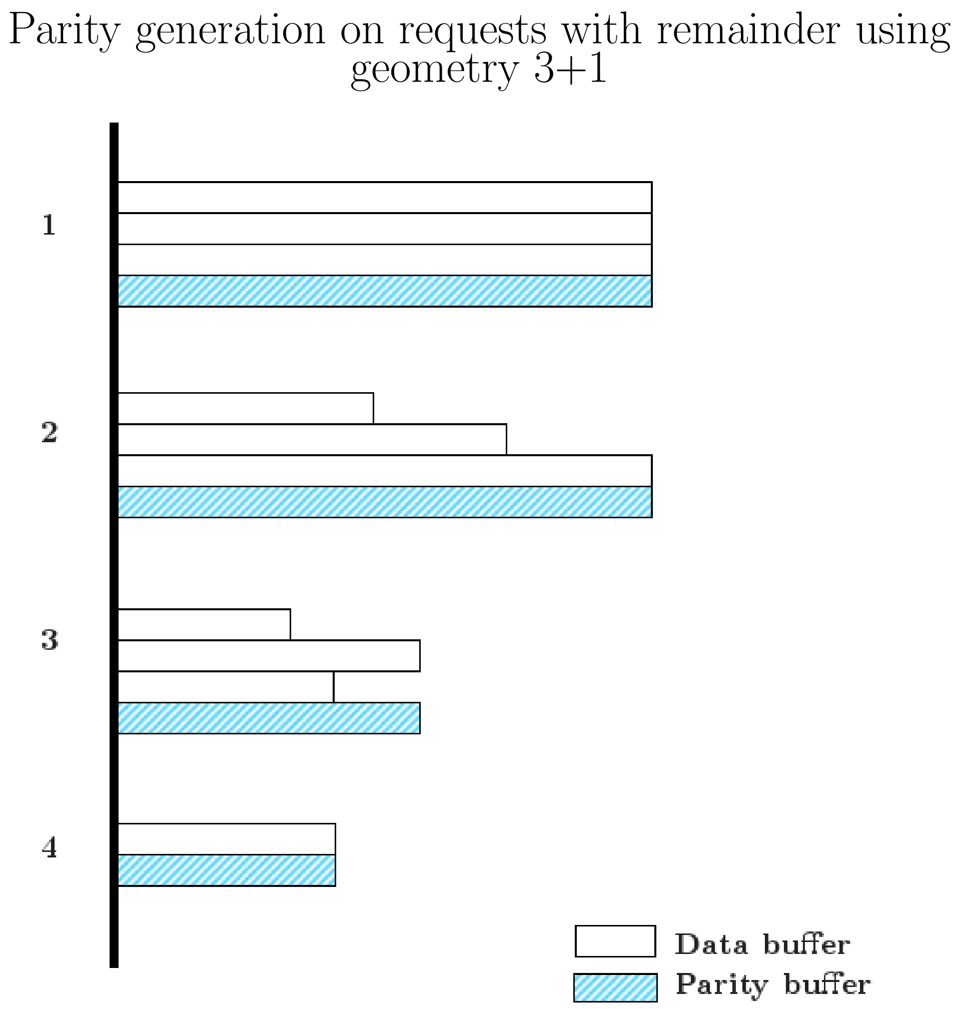
Parity generation on requests with remainder using geometry 3+1
1
2
3
4
Data buffer
Parity buffer

Figure 4.2 Parity is generated based on the size of the packets in the same parity group. The picture shows different cases with different amount of data per packet and different packet per group.

Figure 4.2 shows the parity generation in different cases based on packet layout:

1. Canonical case: packet geometry is complete and every packet completely fills a maximum packet size.

2. Geometry is complete but only a buffer reaches its maximum size and parity is generated based on the biggest buffer, considering zeros for the other missing data

3. None of the buffers reaches its maximum size, so parity is based on the biggest one
4. There are not enough buffers to fill the group and parity is generated based always on the biggest buffer.

4.2 Data Recovery

This is the case when a device cannot read anymore from one of its devices, for any kind of reason. HDD are prone to failures and SSD, the devices used in this system, are more likely to fail over time due to usage.
The server is now unable to satisfy the client's requests but using the *Erasure coding* system, it can recover data comparing packets of the same parity group and its parity.
Parity can be disabled, activated with 1 or 2 parity packet. The more parity we Introduce, the slower the system become, gaining robustness against hardware errors. Whenever a file, or a part of it, is written in a server, the latter propagate its metadata to the next server, in order to protect not only data, using parity, but also metadata of the files.

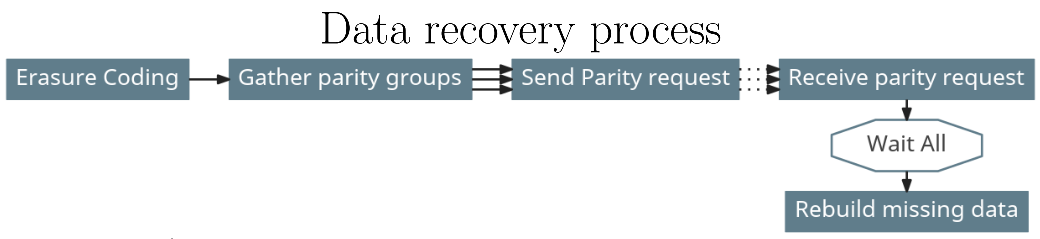

Figure 4.3 If some data is lost, it can be recovered using the Erasure coding system, gathering and comparing data of the same parity group of the packets lost

If the server is not able anymore to read nor data or metadata, it will contact its next server to understand which files are missing. After this step, the server is aware of the parity groups to gather to recover the data. It will then contact the servers that owns the packets of the same parity groups of the missing ones.
After every packet of a parity group is retrieved, the missing buffer can be regenerated using the parity information. If has been lost a parity buffer,

recovery is anyway necessary to prevent future failures.
In the picture the *Gather Parity Groups* is relative to metadata gathering. *Send parity Request* is the request for the buffers of the same lost buffer's parity group not owned by the server. This is a network request, so logically will come *Receive Parity Request* but will happen later in time, after the target server will have sent the answer. After every buffer of the same parity group will be received, the server is able to *rebuild its missing data*.

4.2.1 Queue Issues

Using the buckets system, targets must be decided in this step and then the buffer queue adapts the data to be sent at a different granularity. The parity groups are chosen based on a pseudo-random sequence that should even out the load to the servers.
This system has some issues when some queue is empty: since the targets are not chosen based on the length of the queue, it may happen that some queue is empty and the parity group will not be complete.

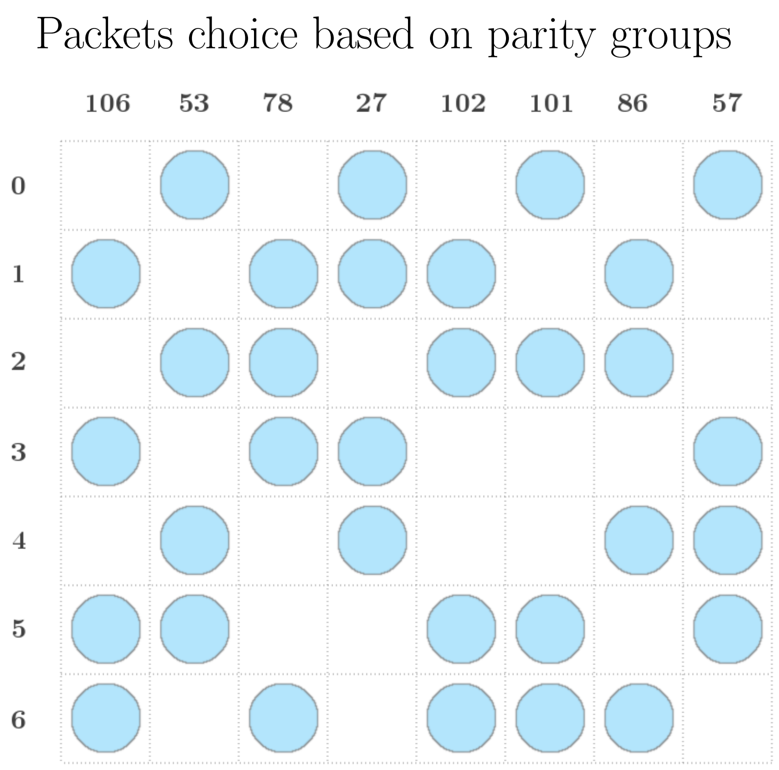

Figure 4.4 Every row represents the queue to a server while every column represents the parity map. A circle means that for the creation of that parity group, a packet from that queue must be popped and sent.

In figure 4.4 each row is a queue while every column shows a parity group. A circle shows that an element should be picked from that queue.
In the first column the parity group identified by *106* shows that should be popped an element from queue *1,3,5* and *6*. The integer identifier is the binary representation of the queue involved. In the first column $(106)_{10} = (1101010)_2$
In this case the algorithm has a bit of tolerance: at the beginning is more likely that a single queue does not have any element so, before dropping this method, it tries some more times to send full parity groups. The chances are in the order of the server count.
After this approach failed more than the chances allowed, another approach is applied: parity groups are sent based on what is left, meaning that the precomputed parity groups are not the only groups inside the system, but new ones can be created based on the needs.

Chapter 5

Performances investigation

After the simulator has been completed and tested, the phase of performances investigation can start. The system has many free parameters that can be set to get a realistic behaviour of the simulator, but also to investigate its behaviour in a different condition.
The following sections will show the performance difference focusing on a single parameter, showing how they will affect the total walltime. For each experiment are done 5 different runs with a different value for that field, keeping also the same request pattern.
The following is the standard configuration used across all the experiments. For every different one a single parameter, or related ones, has been modified.

Field	**Value**	**Description**
TOKEN_COUNT	24	Client parallel communications
GEOMETRY_BASE	3	Data packets per parity group
GEOMETRY_PLUS	3	Parity packet per parity group
SERVER_COUNT	20	Server inside the system
HDD_DATA_COUNT	20	Server Disks per server
HDD_DATA_READ_MBps	2048	Server Disk read performance in MBps
HDD_DATA_WRITE_MBps	1024	Server Disk write performance in MBps
HDD_DATA_LATENCY_us	100	Server Disk latency
HUB_BW_Gbps	80	Network maximum bandwidth
NETWORK_LATENCY_nS	6667	Network latency
NETWORK_BUFFER_SIZE_KB	1024	packet size used for communication
BUCKET_SIZE	8192	grouping size used for file allocation
READ_BLOCK_SIZE	4	linear read block size

5.1 Geometry

In this experiment has been investigated how the geometry affects performances. Geometry is expressed as a pair of number $a + b$ where a is is the

data packet per parity group and *b* is the parity packets per parity group. This is an option inside the system, meaning that parity packets can be used or not based on user needs. The more parity is used the more data loss can be recovered at the same time at the expense of performances. On the other side can be turned off to communicate data as fast as possible aware of the fact that the running experiment can not be recovered in case of hardware failure.
For this experiment has been used the standard configuration with the following settings for geometry

Run	1	2	3	4	5	6	7	8	9	10
GEOMETRY_BASE	2	4	6	8	10	15	20	30	40	50

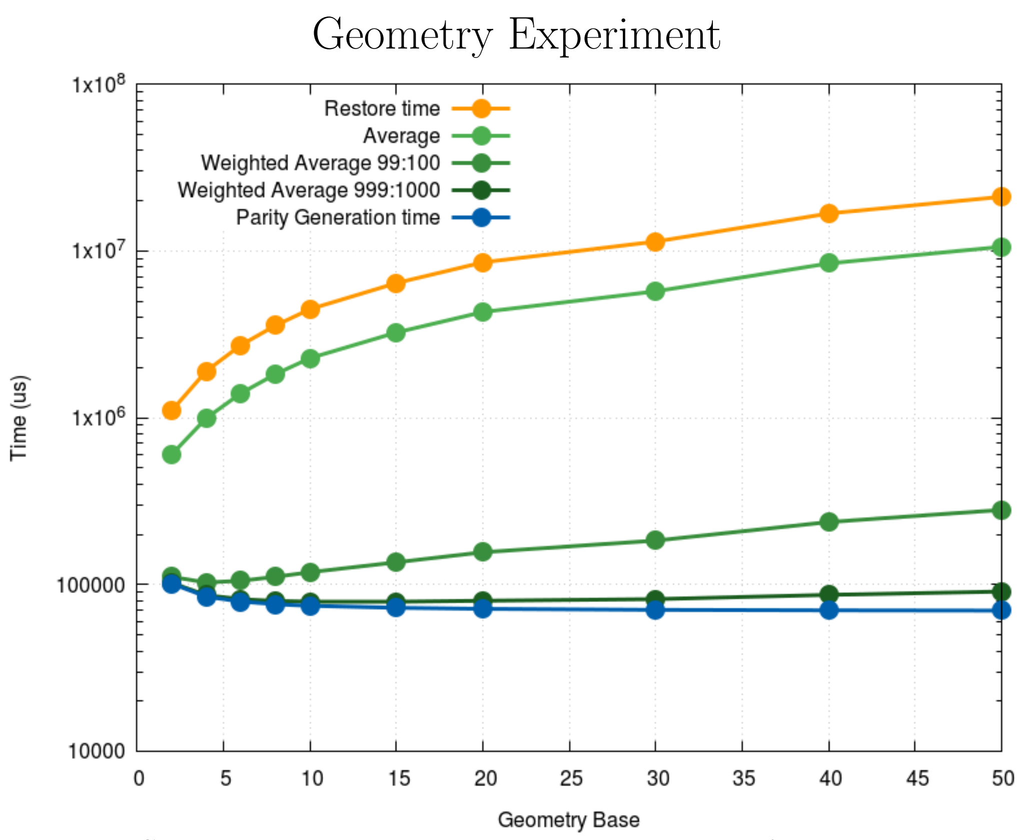

Figure 5.1 Growing the base geometry size reduces the amount of overhead introduced in a system at the expense of recovery time, which keeps increasing

It's clear from figure 5.1 that while the time required to generate parity barely decreases after a base geometry of 10, the recovery time still increases since to recover every packet other $geometry_{base} + geometry_{plus}$ need to be contacted, involving network communication and, in this case, congestion. To better inspect the situation is better to not look at the single values but a value using a formula involving both the measures. Initially has been used a simple average between the two measures, that highlighted that the best size of the geometry was the smallest, since the recovery time increases faster than the parity generation time decreases.
But what actually happens is that parity generation occur much often than recovery, which hopefully should not happen at all. So the simple average has been switched to a weighted average and different weights has been assigned. In the first weighted average parity generation is 99/100 and the better solution shifted from $x = 2$ to $x = 4$.
Using a different weight the solution shifted again to $x = 15$, so in the case where a parity generation happens 99.9% of the time, is ideal a geometry of 15+1.
Keeping increasing the weight, representing a more realistic behaviour of the simulator, suggest to increase as much as possible the size of the packet since is very difficult that the restoration could happen, and in that case, is ideal to wait for it to finish slowly but leaving the entire system with a very low parity overhead.

Geometry	Parity Gen.(ns)	Restore (ns)	Avg 99% (ns)	Avg 99.9% (ns)
2	101083	1103562	111107	102085
4	84512	1904740	**102714**	86332
6	78900	2723940	105350	81545
8	76173	3572632	111137	79669
10	74507	4476210	118524	78908
15	72508	6415274	135935	**78850**
20	71469	8532990	156084	79930
30	70440	11373276	183468	81742
40	69993	16756538	236858	86679
50	69738	21087047	279911	90755

5.2 Disk Bandwidth

The technology involved in the servers is not the most powerful so the servers can be improved. The standard settings in the simulator are close to the performances of an SSD, reading speed at 2GB/s and writing speed at 1GB/s.

The usage of NVMe memories or PCIe ones can lead to better disk performances having a lower latencs and removing the bottleneck of SATA protocol. This experiment run the tests with the following settings

Run	1	2	3	4	5
DISK_WRITE	512	1024	2048	3072	4096

In this case the read bandwidth is supposed to be twice the write bandwidth.
To better investigate the influence of the disk bandwidth on the final time, a setting with a more narrow bandwidth has been tested. If the results are acceptably bad, IME could be designed to have cheaper drives and invest more on more meaningful features. Is to consider that has been changed only the maximum bandwidth in these experiments so every other measure such as latency will not affect the final results.

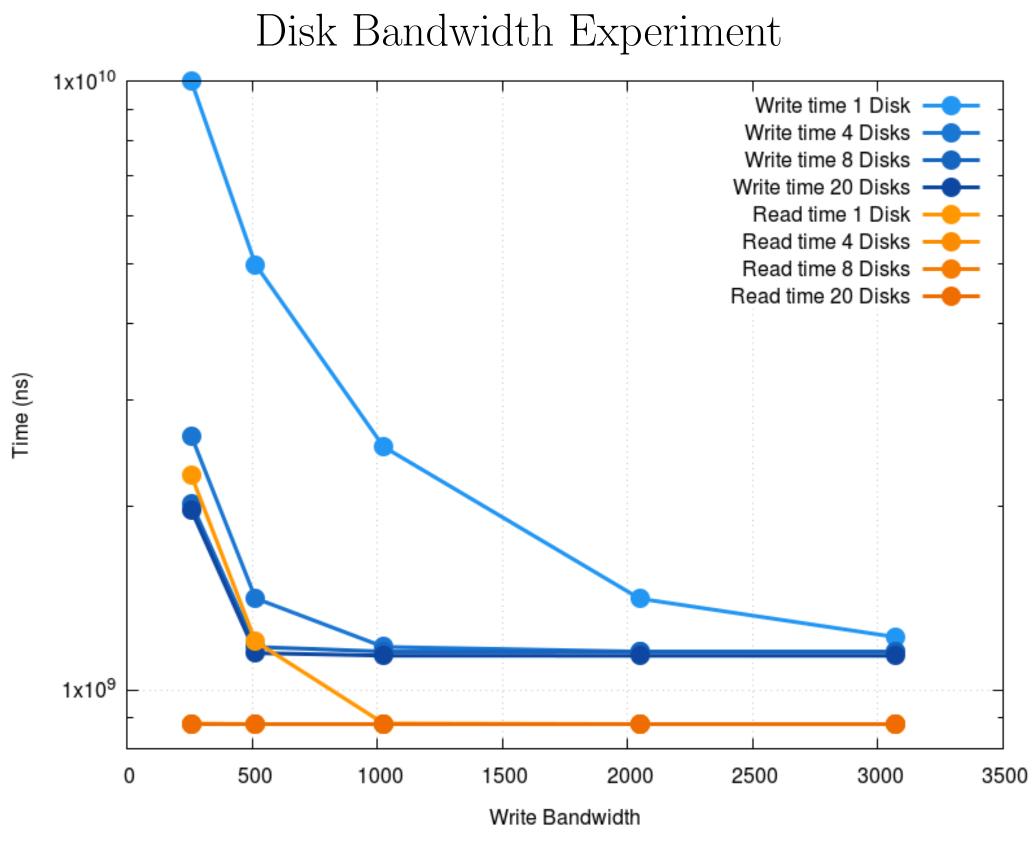

Figure 5.2 Disk bandwidth R/W wall times using a different disk quantity per server

---

Figure 5.2 shows that the single disk performances really depends on the configuration installed in a single server. In the extreme case where a single disk is installed per server, single disk performance really matters, being able to reduce by a factor of 5 the total walltime for the write operation, as the *Write time 1 Disk* shows.
Nonetheless a real world server uses 20 disks per server and in this case with such a parallel environment the single disks performance seems to not matter at all. What happens in this case is that CML_oid s are stored in different disks, so for such a small transaction is more important the disk latency than the disk bandwidth, and that factor in this experiment has not been modified since this should involve a different technology instead of a different SSD drive.

5.3 Server count

In this experiment the aim is to show which benefits can be obtained increasing the number of servers. The aim of DDN is to be able to make use of thousands of servers, being able to satisfy a lot of requests at the same time.
The setup used in the experiments are the following:

Run	1	2	3	4	5
SERVER_COUNT	4	16	64	256	3000

A large amount of servers are useful to satisfy many more requests at the same time, but they are not useful in case the clients involved in the simulation are too few, even if they have a large amount of data to communicate. The firsts runs failed to show any improvement because of too few clients involved in the simulation. More clients has been instantiated communicating less data at the same time.
The following are the results of the simulation with a different request pattern

Multiple tests has been executed using different configurations. The 2 experiments shown in figure 5.3 are completed using a request of total size 256GB and 8GB.
The test in this experiment are more close to a benchmark than a real case simulation: since both read and write operation involved the whole dataset, to better compare the recover operation also the latter has been performed over the whole dataset. This is not possible in the real world because the system could not survive to a complete disk failure at the same time, but for simulation purposes is fair to compare different operation on the same

dataset.

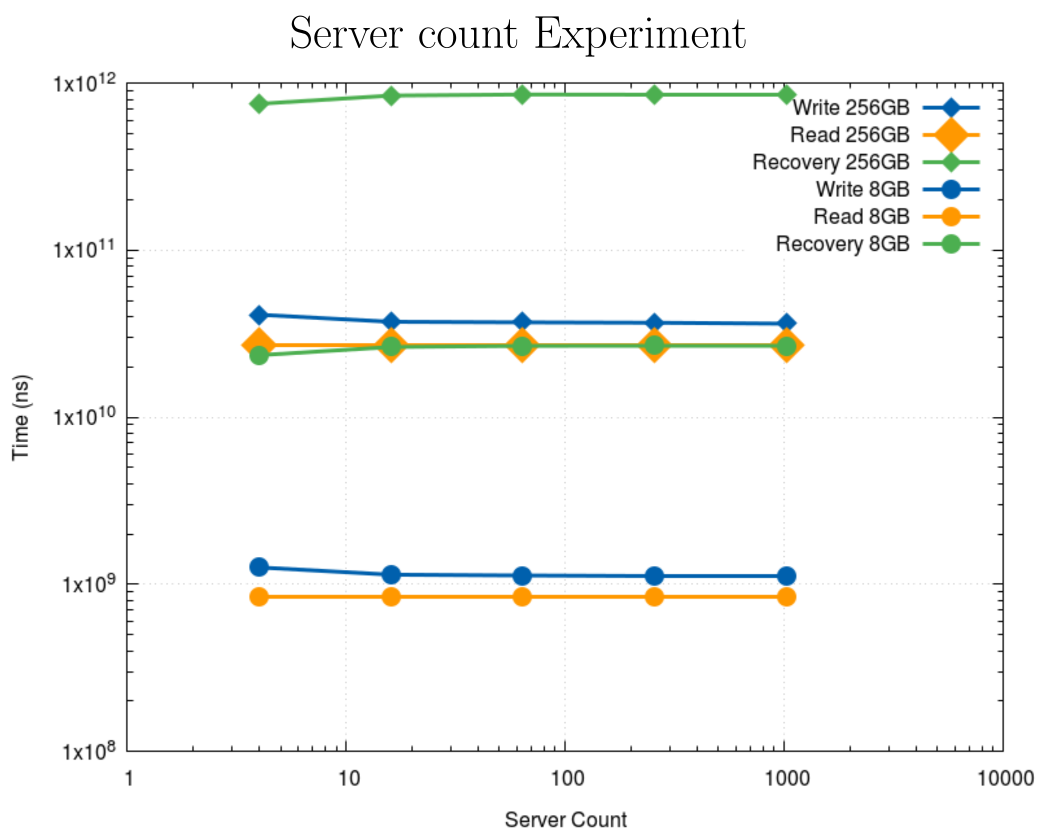

Figure 5.3 The number of servers seems to not affect much the performance improvement. At the current time they only allow a bigger total disk capacity

As figure 5.3 shows increasing the number of server does not increase drastically the overall performances. The experiment about the disk bandwidth in section 5.2 showed that the time required for the I/O operations takes now a small percentage of the total simulation wall time. The real bottleneck in this situation is the network bandwidth: more requests take more time because of data transfer from client to server and vice versa. To figure out the load of data to be written from each single disk can be computed arithmetically. Let

$$d = initial\ data$$

$$D = total\ data$$

$$g_b = geometry\ base$$

$$g_+ = geometry\ plus$$

$$s = total\ SSDs$$

$$w = SSD's\ write\ speed$$

From simple arithmetic the total data D and total time T required for write operations are

$$D = \frac{d(g_b + g_+)}{g_b}$$

$$T = \frac{d(g_b + g_+)}{g_b s w}$$

From this formula can be computed the following data:

Init data(GB)	Geometry	Total SSDs	Write Speed(GB/s)	Total Time(s)
8	3+1	8	0.15	0.89
8	3+1	80	1	0.13
8	3+1	320	0.15	0.22
8	3+1	320	1	0.03
8	9+1	80	0.15	0.74
8	9+1	80	1	0.11
8	9+1	320	0.15	0.19
8	9+1	320	1	0.03
256	3+1	80	0.15	28.44
256	3+1	80	1	4.27
256	3+1	320	0.15	7.11
256	3+1	320	1	1.07
256	9+1	80	0.15	23.70
256	9+1	80	1	3.56
256	9+1	320	0.15	5.93
256	9+1	320	1	0.89

Despite in the real simulation network traffic and some more mechanics makes the data more variable, we can detect a baseline here. The parameter has been chosen as follows:

- Small and big transaction to show how much the transaction size matters

- Different geometry to show the influence of geometry
- Different server count to use a more realistic server configuration
- A write speed comparable to an HDD to show the importance of SSDs in this technology

The most realistic configuration is represented by the last line showing that with a transaction relatively big, still the IO operations takes few time compared to the total simulation wall time as show in figure 5.3.
To really make use of a big amount of servers, network should scale as well as servers quantity, at least as total transaction time will be comparable to the I/O times.

5.4 Conclusion

In this thesis we have addressed the problem of performance evaluation for an existing system and as well performance estimation for non-existing system. The method has been applied to an industrial Burst Buffer technologies but can be re-targeted to other component of HPC cluster.
The critical point in the performance evaluation of the complex system, such as Burst Buffer, is the existence of multiple threshold in the performance equation. Depending on the quality of the devices, their quantity, the network capabilities, the workload a different bottleneck may be the limiting factor. This is why engineers need to use such tool in order to quantify and to balance their system.
During the thesis, additionally to the reverse engineering of the IME Burst Buffer, important effort has been dedicated to the design and implementation of a discrete event simulator. Such kind of simulator has been chosen after a review of the existing trade-off of simulation technologies. We have executed validation of the simulator on a per component basis. In term of system integration as the ability to simulate the whole system, we have executed functional tests but fall short for comparing our simulator with real measurement on IME Burst Buffer.
Two issues have been preventing us to complete this task:

- The component dedicated to the simulation of the network is too simplistic: currently the network topology is considered as a single bus shared among all the clients and servers while IME network is more close to a full fat tree. This decision was taken to in order to meet the deadline and show function validation of the full simulator at the cost of its accuracy. We plan to complete and demonstrate a more realistic network component by the time of the defense.

- The execution time of the simulator has not been optimized enough. Again this was a decision taken to meet the goal; of a functional simulation of the full scale system. Anyway it appears that simulating a complex workload on a large scale system may be prohibitive and require hours of execution time. The simulator is still has only a serial implementation, not taking advantage of a multi-core machine.

Acknowledging these two limitations we consider that the designed simulator can be helpful in order to guide developers and storage architects towards a more balanced and scalable system, bringing some value to the community.
From these 6 months work we think that an important point, if not critical, is the ability to build a system as a set of articulated components, where components can be fed by micro-benchmarks measurements from real systems. Our work includes two examples with the network performance equation and the parity computation.
As future work we would like to address first a finer description of the network in order to address the balance between the number of clients and server depending on the workload. A longer term opening would be the ability to take as an input and simulate I/O traces, either synthetic or collected on real life systems.

Chapter 6

Glossary

In the main text every topic is explained before it is used implicityl in the context. Anyway it is easier to dedicate a section with a small explanation of every detail.

Buffer: short way referring to a *Network Buffer.* This is a packet of 1 MB of data to be sent or received. From external works we know that this size is the most suitable to maximize Infiniband performances

CML_oid: inside a server, the single packet of data received is split in CML_oid, at byte oriented level, and then written to different devices.

Device: in general a device refers to a server's SSD

Diagonal Limit: analytical model to simulate the behaviour of the network or disks bandwidth. Explained in detail in section 2.1

Eager Commit: in the server context, metadata propagation is cached to send as less network communications as possible. If a client has finished transmitting a file to a server, it will send the last piece of data as an eager commit, forcing metadata propagation

Geometry: this measure defined as $D + P$ represents how a parity group is constructed. D represents data packets and P parity ones. See section 4.1.2 for more details

Metadata propagation: in the server context, in order to keep a backup in case of drive failure, metadata is sent periodically to other servers

Parity Group: in erasure coding context, a request takes part of a Parity Group. If a single request is corrupted, the others can recover it based on the additional information, the parity

Parity Map: given a packet, it has to keep track of its own parity id and the location of the others packets belonging to the same parity group. This information is the parity map

Parity Request: after data corruption, the server asks for the packets with same parity group id of the packets lost to proceed with recovery

Token: Before starting a packet transmission, the clients need a token. The token is consumed upon the transmission start and recovered after the operation being completed, received the acknowledgment from the server

Bibliography

[1] Dot language. https://en.wikipedia.org/wiki/DOT_(graph_description_language).

[2] Python threading module. https://docs.python.org/3/library/threading.html.

[3] Token ring. https://en.wikipedia.org/wiki/Token_ring.

[4] Jean-Thomas Acquaviva. Data protection and erasure coding. https://drive.google.com/file/d/1XFAEAXGlwP_-9AHuDNGwpzjS5TVUO3vU/view?usp=sharing.

[5] Gregg Brendan. Flamegraph. http://www.brendangregg.com/flamegraphs.html.

[6] Team Simpy. Container tool. http://simpy.readthedocs.io/en/latest/api_reference/simpy.resources.html#module-simpy.resources.container.

[7] Team Simpy. Discrete event simulation for python. https://simpy.readthedocs.io/en/latest/.

[8] Team Simpy. Resource tool. http://simpy.readthedocs.io/en/latest/api_reference/simpy.resources.html#module-simpy.resources.resource.