Turkish Journal of Electrical Engineering & Computer Sciences

http://journals.tubitak.gov.tr/elektrik/

Research Article

Turk J Elec Eng & Comp Sci
(2022) 30: 1571 – 1585

doi:10.55730/1300-0632.3866

Lightweight distributed computing framework for orchestrating high performance computing and big data

Muhammed Numan İNCE*, Melih GÜNAY, Joseph LEDET
Department of Computer Engineering, Akdeniz University, Antalya, Turkey

Received: 20.10.2021 • **Accepted/Published Online:** 08.05.2022 • **Final Version:** 31.05.2022

Abstract: In recent years, the need for the ability to work remotely and subsequently the need for the availability of remote computer-based systems has increased substantially. This trend has seen a dramatic increase with the onset of the 2020 pandemic. Often local data is produced, stored, and processed in the cloud to remedy this flood of computation and storage needs. Historically, HPC (high performance computing) and the concept of big data have been utilized for the storage and processing of large data. However, both HPC and Hadoop can be utilized as solutions for analytical work, though the differences between these may not be obvious. Both use parallel processing techniques and offer options for data to be stored in either a centralized or distributed manner. Recent studies have focused on using a hybrid approach with both technologies. Therefore, the convergence between HPC and big data technologies can be filled with distributed computing machines at the layer described. This paper results from the motivation that there exists a necessity for a distributed computing framework that can scale from SOC (system on chip) boards to desktop computers and servers. For this purpose, in this article, we propose a distributed computing environment that can scale up to devices with heterogeneous architecture, where devices can set up clusters with resource-limited nodes and then run on top of. The solution can be thought of as a minimalist hybrid approach between HPC and big data. Within the scope of this study, not only the design of the proposed system is detailed, but also critical modules and subsystems are implemented as proof of concept.

Key words: Distributed and parallel computing, big data, high performance computing, distributed programming, resource management

1. Introduction

After 2005, the need to process batch data generated over the internet in real-time and the recent flood of genomics data created the necessity for high performance computing systems that scale, are easy to distribute and implement, and yet accurate at the same time. Although there exist HPC technology stacks and researchers, less experienced researchers did not rely on them and created their own parallel computing stacks such as Hadoop and Spark. This is perhaps because the HPC community did not wish to invest time and resources to alter existing HPC toolsets to accommodate new comers. Instead, the new comers were expected to alter what they were doing to fit existing HPC toolset.

Hadoop [1], the dominant big data management technology stack, came to market with the claim of being able to run on commodity hardware within a cluster setup. But, in practice, Hadoop-based projects deployed onto cluster nodes with increasingly required CPU and RAM resources. Today, we attempt to solve

*Correspondence: muhammednince@gmail.com

large analytical problems that use big data and require high performance computing at the same time. While Spark's RDD (resilient distributed datasets) is preferred over Hadoop/MapReduce due to its performance and ease of use, when compared with OpenMPI it still underperforms. Additionally, in our previous work [2], we attempted to make a Hadoop-based distribution. However we experienced many challenges from deployment to configuration. It did not run using reduced resources on commodity hardware as claimed or expected and was a resource-hungry component.

High performance computing (HPC) is an area with different parallel programming and shared memory techniques and approaches that solve big computational problems. On the other hand, the concept of big data is based on collecting, processing and analyzing the data produced by social media, business and IoT sensors. While HPC systems focus less on data management and more on designing high-performance algorithms, big data systems concentrate on effective data management, data inquiries and real-time streaming applications. Parallel computing in distributed systems for big data management implement a kind of divide-and-conquer algorithm such as MapReduce, whereas the HPC systems implement parallel processing and messaging through protocols such as MPI (message passing interface). Programming APIs and data abstractions are quite different between the big data and HPC systems. HPC systems have adapted low-level APIs, while big data systems have adapted high-level user-friendly APIs. Performance and usability are a delicate balance for both systems, and achieving performance while maintaining usability is a challenging job. On the hardware side, while HPC systems require high-speed network service such as InfiniBand, Big Data Management software primarily relies on solutions where data and computing are done on the same nodes.

Meantime, cloud-based solutions are based on collecting requests (data/computation) at centralized nodes and distributing it to powerful machines over the Internet without requiring access to local servers. This metaphorical cloud concept is developed due to the gap between HPC and Big Data. However, in order to keep this infrastructure alive, internet access or full-featured network devices are required. Furthermore, with the increase in the amount of data collected for Business Intelligence or Science, the logarithmic increase of the cost of cloud-based services became a burden and a limiting factor.

Because of the need for a low cost, high capacity, high performance computing system, the proposed design and the proof-of-concept implementation of a light-weight distributed computing research framework with the following novel features are introduced:

1. Each board with Linux of different architectures can be located in the compute pool as a cluster under low resource consumption.
2. A flexible network layer in which nodes can create smart network patterns inherently in favor of computing.
3. Programmable modular configuration system for resource negotiation and different computing techniques (without XML or shell scripts).

The above features highlight the general characteristics of the framework. Due to the increasing number of options of computational machines such as IoT devices, smart phones, tablets, desktop computers and servers, the internode communication infrastructure with a lightweight resource/cluster management system requires attention. Such communication infrastructure has to be minimally complex for communication yet support the smart network topology for Internet protocol. Therefore, in this study, we aim to take on this need by revealing the state of the art in the distributed and parallel computing space.

Parallel and distributed data storage and processing frameworks are often used in handling large volumes of data and generating analytical results for science and businesses. First HPC (1980s), then Hadoop (2000s)

implemented these concepts with very different approaches. Recently, hybrid solutions have come to the forefront as service-oriented architectures that easily integrate with each other and thus support smoother data exchange. This is why advances are seen in both the Big Data and HPC technology stacks brought by these advantages. HPC technology primarily focuses on high-performance hardware and low-latency networks to process mostly structured data running scientific algorithms. On the other hand, Hadoop for Big Data applications processes Large volumes of unstructured data on the commodity hardware with data parallelism. Therefore, in order to process Big Data with high performance then HPC and Hadoop has to inevitably converge. This convergence is also observed at the component level and now the components have started to offer mutual support. We discuss the component-based assessment in the following subsections that helps us to determine the features that are required for a framework that bridges Big Data and HPC.

1.1. Distributed file systems

A distributed file system that splits data/resources physically or virtually to nodes is at the core of architecture. In this structure, the file is split into chunks and may be located at any node. The complexity of splitting and locating data is hidden from the end-user often through an interface managed by the master node. These nodes may reside on private (Intranet) or public (Internet) cloud. Among the distributed file systems, i) Hadoop (HDFS), ii) Lustre, and iii) Gluster stand out.

Hadoop (HDFS) has a master/slave architecture and is designed to run on commodity hardware with high fault-tolerance. HDFS performs well in accessing and retrieving large data sets. **Lustre** offers high performance through parallel access to data and distributed locking. Lustre has a POSIX-compliant namespace that is generally rock-solid for large-scale computing infrastructures. **Gluster** is a scalable distributed file system that expands and provides additional storage as needed. Gluster is less complex as it does not depend on metadata server.

In a distributed file system where a single NameNode exists, it may limit scalability and become a resource bottleneck and a point of failure. If it fails, the file system becomes inaccessible and when it comes back up, the name node must replay all the operations in queue. For large clusters, this replay operation may take hours. Although Hadoop promotes data locality, this is not very realistic in a heterogeneous network. Lustre is designed mostly for HPC environments and requires high speed internet and meticulous work for management. In heterogeneous networks, however we usually do not have Infiniband like network connections. GlusterFS may be preferred as the distributed file system base for the proposed architecture for the following reasons;

- Easy to manage and independent from kernel while running in user space.
- Improves the performance of data and objects by eliminating metadata server.
- Easy to deploy and run on SoC boards like Raspberry PI.
- It does not need an intermediary server. Clients may directly mount the block device.
- Translator mechanism exists for extensible functionality.

1.2. Parallel programming models

There are several parallel programming models which allow to define locality and concurrency: threads, message passing, data parallel, single program multiple data (SPMD) and multiple program multiple data (MPMD) models. These models are used for abstraction above hardware and do not tie to a particular computer architecture. Next, we discuss each model and their implementations with respect to distributing computing.

Threads have two distinct implementations; Pthreads and OpenMP. Pthreads often come with the C programming language's built-in thread library (libC). They require significant programmer attention to properly implement explicit parallelism. OpenMP has a shared-memory multiprocessing API and provides several compiler directives for creating threads, synchronizing tasks and managing the shared memory on top of Pthreads. Pthread (POSIX Thread) and OpenMP (Open Multi-Processing) APIs are used widely for parallel computing. OpenMP APIs are more convenient in code for splitting the execution using directives. Therefore, it is more appropriate to be used in a framework.

MPI is a message passing library standard based on the consensus of the MPI Forum including users and developers. MPI aims to establish a portable and sufficient standard for messaging in distributed programs. In MPI, each process communicates with each other by using its own address space. Partitioning workload and delegating tasks of the workload is facilitated by MPI. Tasks use local memory, however computation is either on the same machine or on remote machines. Data related to communication information is also exchanged among tasks. MPI also provides library functions to manage message passing in blocked and unblocked modes.

MapReduce has been a pioneer model for large-scale parallel batch data processing that uses key/value storage. An index mechanism is defined over the keys to speed up arbitrary search. The core idea of MapReduce comes from a functional programming approach that uses main Map and Reduce functions. Both functions have two input parameters, a set of key/value pairs as an input dataset and a user function. Map and Reduce processes the data with the user-defined function often recursively on subsets [7]. The MapReduce runtime considers data locality during launch and the programmer does not need to worry about data partitioning, process creation, and synchronization. The same Map and Reduce functions are executed across machines, therefore the paradigm may be regarded as a kind of SPMD model [4].

For the proposed parallel computing model, OpenMPI is more appropriate as; a) it is the de facto standard protocol for all HPC platforms, b) does not require additional coding for portability, c) supports vendor specific native algorithms for higher performance, d) there exist many applications from both vendors and public sources, e) provides essential virtual topology, synchronization, and communication functionality between nodes.

MPI is also ideal for iterative algorithms where nodes require data exchange between processors to proceed. In general, MapReduce is more preferable for noniterative algorithms where nodes need reduced data exchange. However, Mapreduce is more recently replaced by more versatile data processing technologies such as Spark and Flink [12]. In our design, we consider a hybrid structure where different technologies may be used based on the requirements. It may be determined at job creation while matching the capabilities of the compute nodes. These procedures may also be specified in the extensible configuration system.

1.3. Resource manager and job scheduler

Resource management and job scheduling are two separate but related problems. By resource management, we refer to the negotiation of resources that are required for running a job. Once the resources that are available for the job are identified, the scheduling is performed by reserving these resources and lunching the job. Those jobs whose resources are yet not available, are enqueued according to the policy of the workload manager [9]. On the other hand, at the application level, most of time is spent at processing significant amounts of input/output data as part of job scheduling [10]. Next, resource management and scheduling solutions for Big Data and HPC are reviewed.

Slurm is an open source and highly scalable cluster management/job scheduling system for Linux.

Slurm runs at user-space and self-contained. For big data, **YARN - Yet Another Resource Negotiator** [13] provides two key functionalities: global resource (ResourceManager) and application (ApplicationMaster) management. The ResourceManager is a master that controls the nodes of Hadoop cluster through NodeManager and allocates system resources to applications while scheduling tasks [12]. **Mesos**, first developed by Berkeley, allows management of the data centers as a single server by abstracting the resources such as CPU, RAM, and Disks on nodes. The other widely used resource manager, **Mesos**, shares resources in a fine-grained manner, allowing frameworks to achieve data locality by taking turns in reading data stored on each node while supporting the sophisticated schedulers of today's frameworks (Hadoop, Spark, Kafka, Elastic search) [14].

Major resource managers that are widely used are discussed in conjunction with the needs of the proposed framework. On the HPC side, Slurm provides a quality solution for the resource management. Since HPC is concerned with a better quality algorithmic solution of the problem, job management has gone on solving resource management jobs with other applications. Of course, this situation caused a more complicated configuration system for the structure like Slurm. YARN runs on Hadoop therefore requires nodes to have Hadoop setup. It supports components such as Spark, Flink, and even HPC/Slurm [15]. On the other hand, YARN is a Java application that runs on JVM. While JVM brings portability, it demands nodes to have additional capacity which may be a limiting factor in introducing thin clients. Therefore, for a network with heterogeneous participants, resource inefficiency leads us to prefer options other than Hadoop. Mesos manages large collections of heterogeneous resources using Linux's kernel-level isolating feature (Cgroup) for processes. Because, Mesos's learning curve is quite steep, it requires custom frameworks in most cases. It should be noted Mesos does not provide all the desired features and may require additional software installations for nodes.

Due to limitations and complexities of existing solutions, we implemented our own lightweight and simple resource negotiator and job scheduler. The simplicity allows us to extend capability with a plugin system supported by a scripting language. The network that we aim to run the system on will have storage/compute nodes with heterogeneous resource composition. Nodes have ability to share their resources based on the job requirements using the scheduling plugins developed. Having two-level scheduling approach, allows system to execute different rules for each workload running on a common resource pool. This approach increases resource utilization.

1.4. Checkpoint/restore tools

Application checkpointing is a facility to save the complete state of an executing program to a disk periodically and in failure, resume the job by restoring the state from the last checkpoint file. The benefits of checkpointing in the HPC environment are immense since HPC jobs may run for several days/weeks. Big data frameworks also implement their checkpoint components but mostly for data restore as backing up JVM is expensive [17].

Berkeley Lab Checkpoint/Restart (BLCR), a system-level checkpoint/restart implementation for Linux clusters, targets HPC applications that support MPI. System-level checkpointing focuses on preemption and making it suitable for responding to fault precursors such as elevated temperature from sensors [18]. DMTCP (Distributed MultiThreaded CheckPointing) is a software for applications that automatically handles operating system artifacts (Threads, I/O, File, Mutexes, etc.). DMTCP does not use kernel functions and it runs unprivileged on user-space. It may be used as a checkpoint-restart module within larger packages [19]. Checkpoint/Restore In Userspace (CRIU) is an application checkpointing solution [20]. It has since been integrated into multiple container runtimes to support container migration. One of the goals of CRIU is to be as transparent as possible. Therefore, CRIU does not require the processes to be prepared in anyway and restores it

with the same PID before the failure. A container or an application can be frozen by CRIU, then it prepares the checkpoint to store it into disk. The stored data can then be used to restore and resume the process. This function provides us to use snapshots for applications and container migrations.

Application-level solutions for checkpointing and fault-tolerance, are considered to be more time and space efficient than system-level approaches. This is because system-level solutions cannot utilize application-specific information [18]. Comparisons of tools are given below:

- DMTCP & CRIU do not need to recompile: useful for legacy/proprietary applications.
- CRIU & BLCR do not need to start anything before the application.
- DMTCP & BLCR can checkpoint MPI.
- CRIU can checkpoint Docker & LXC containers.
- Small overhead with CRIU & BLCR.

Due to advantages of various approaches, we suggest a flexible checkpointing design that may adapt CRIU for Linux containers and Docker compatible operations and DMTCP for MPI.

1.5. Communication

Slurm provides a REST API daemon for communication. Mesos currently uses an HTTP-like wire protocol to communicate with its components that uses asynchronous libprocess library. Libprocess primitives have message passing actor and its messages are immutable that makes paralleling easier. YARN requires its own abstract RPC protocol to be implemented by the frameworks to connect various components. This brings extra complexity in protocol design. In YARN, between any two components that needs to communicate via RPC, one end is a client and the other end is a server. The client always actively connects to the server. Therefore, YARN actually uses a pull-based communication model. Also YARN RPC does not have a flexible API for different programming languages. However, it provides access through the thrift protocol, which is slower than the native access.

For a fully distributed computing platform that includes IoT devices, the communication layer should be simple and less feature rich. At this point, ZeroMQ [21] provides a crossplatform transport that carries messages across inproc, IPC, TCP, UDP, TIPC, multicast and WebSocket. It is trivial to add an http-to-raw converter plug-in if it needs to reside behind a NAT (network address translation). In some cases, where bandwidth is limited and performance is important then going with a flexible socket construction is the right choice. ZeroMQ handles all the complexities of managing connections and queuing data for unreliable connections. Data between nodes may be routed internally by ZeroMQ so that bidirectional communication is provided. ZeroMQ focuses on keeping the payload overhead efficient and provides scalable communication models such as KQueue or EPOLL on Linux.

Our paper is organized as follows; after the necessary components for the framework have been critically discussed in Introduction, Related work is compared with the proposed design in Section 2. Section 3 presents the design of the framework and the components chosen. The proof-of-concept implementation called **BasatNet** is introduced in Section 4. Experimental results are presented in Section 5 and finally the conclusion and future work is discussed in Section 6.

2. Related work

From a high-level perspective, YARN and Mesos may be considered a close alternative to the proposed project for the Big Data tasks and Slurm for the HPC. But these alternatives are not light-weight and difficult to be

made autonomous. For example, YARN is a resource manager specific to Hadoop. On the other hand, Mesos manages cluster through Zookeeper or other resource managers [31]. Slurm is not suitable for big data as a workload manager as it requires detailed configuration. We mostly discussed the projects that can be pioneers for Big Data and HPC and will provide convergence between the two. These projects are reviewed below and compared also in Table .

Flux [22] is a framework that consists of a suite of projects, tools, and libraries that may be extended to build site-custom resource managers for HPC. Flux also has an improved scheduler for highly scalable workflows [23]. The Flux overlay network implements cryptographic privacy and end-to-end data integrity using ZeroMQ TCP connections. Unfortunately, Flux only targets the HPC domain and requires integration with Slurm.

Twister2 [25] has a framework structure that addresses both Big Data and HPC needs by including; Single/Multiprocessor CPU capability, nonuniform memory access support, multistorage units, high performance network support (Infiniband). Even though Twister2 appears to target HPC environments with OpenMPI [25], it has an abstract deployment method where the cluster type needs to be specified for the jobs that are submitted. Therefore, it requires a load manager such as Slurm or Mesos [26]. Consequently, Twister2 does not take the responsibility of a cluster management in the infrastructure and plays a role at the toolkit level. Its dependence on an existing framework introduces additional complexity.

HPC-ABDS: High Performance Computing Apache Big Data Stack proposes a middleware solution that combines the best of Apache Big Data and HPC Stacks as two separate software packages [27]. HPC-ABDS work began as the aforementioned Twister2 project [24] fork. HPC-ABDS has attempted to define a qualified intermediate layer that serves as a bridge between the two stacks. This project led to the birth of MIDAS/SPIDAL.

MIDAS/SPIDAL: Big Data applications have a wide range of features that require different programming models and different forms of parallelism, from execution of many loosely coupled tasks to in-memory caching and parallelism for iterative MapReduce. Thus, MIDAS provides a scalable run-time system, the underlying resource management middleware and a heterogeneous infrastructure access layer for efficient operation across applications. The successors of this study have evolved towards the development of a SPIDAL library project [28] that links HPC and Big Data instead of reducing it as it is. While this was considered reasonable 10 years ago, it is now insufficient for the growing mass of big data applications as it not clear how different applications will integrate with each other. Our study differs from this study in that advocates a lightweight design features such as running on commodity hardware.

3. Design and architecture

The basic components of a distributed computing environment were discussed in the background section. Components that are selected should be lightweight yet efficient in computation. In this section, the design groups and implementations are separated functionally and examined. Design groups include Submission, Coordination, Computation, Communication and Data. Some of these groups may have existing solutions or their own implementations. Considering the features that are listed in Introduction, **BasatNet** is designed to provide fault-tolerant coordination with built-in powerful distributed messaging. The interaction of groups is depicted in Figure 1 and detailed below.

Table . Feature comparison table of frameworks.

Feature	**Flux**	**Twister2**	**HPC-ABDS/MIDAS**	**BasatNet**
Domain	HPC	Big Data/HPC	Big Data/HPC	Big Data/HPC
Language	C	Java	Java	C
Distributed Data Mgm.	Lustre	HDFS/Spark RDD	HDFS/Lustre	GlusterFS/NFS
Resource and Job Mgm.	Fluxion(flux-sched)	Slurm/Mesos	Slurm/Yarn	Basatctl
Coordination	Own	Zookeeper	Zookeeper	Basatd
Parallel Computing	PMIx	Spark/Flink/MPI	Hadoop/MPI	MPI/OpenMP
Communication	ZeroMQ	Conv. MPI, Harp	Kafka/RabbitMQ	ZeroMQ
Commodity Hw. Operability	Yes	No	No	Yes
Configuration	Toml	XML	XML	Lua
Checkpoint	Slurm	Chronos	-	BCLR/CRIU

Submission of a computing job from a client node to submission controller is shown in Figure 1, submission block. A client application such as basatcli, for example, may submit a computing task with desired settings either by command line parameters or using a configuration file. Basatcli uses several URL schemes to allow different strategies for the dissemination of code execution. These schemes may include HTTP(S) and a local mount protocol that also involve SSHFS, NFS (network file system), or Libgfsapi(GlusterFS). Transmission of the computing task ideally aims to provide the minimum network latency delay. Submission also involves preparing and submitting the configuration to the controller. BasatNet uses a programmable configuration system for smarter execution. Lua scripting language is used to manage configurations due to being lightweight and convenient for coding settings. A Lua configuration file generally contains the necessary settings to ideally distribute the tasks of an application. These include resource allocation such as CPU, memory, I/O, parallel computing techniques. All these adjustments may be managed by the Lua programming language.

Service discovery is widely used in distributed applications and describes the ability of distributed services to find each other using service names without knowing the physical location and the address of the server. It is important that the services be discovered easily. In our design, service discovery is a dummy transmitter. Basatnetd (Basat network daemon) acts as a router and is used to maintain robust synchronization between nodes that are distributed. Basatnet keeps track of the status of worker and supervisor nodes with heartbeat and service-oriented request-response processing. The Basatnetd atomic broadcast protocol is at the core of the whole system that issues orderly updates. Basatnetd uses the server-side service discovery in which clients make requests. Consequently, the service registry is queried by Basatnetd and a request is forwarded to an available instance.

Coordination and management of the distributed computing is critical as fundamental tasks such as heartbeats, resource allocation, job management, checkpoint tracking and monitoring is done via coordinator controller. The software acts as a resource negotiator and in case of a failure, it seeks new resources via

dynamic allocation. The Basat controller (Basatctl) in our design also includes job management integrated with resource allocation. Basatctl creates compute pool groups from agents that connect to itself. By default there is always at least a single computation group available. Various resource groups may be allocated for special computations. Agents report their computing environment capacity such as CPU, memory, I/O capability to the controller node. The controller distributes the job by taking into account the agents capacity and the submission configuration. Compute pools may support different job scheduling algorithms. For proof of implementation, a FIFO scheduling algorithm was chosen. If the controller detects the job is finished when all nodes return their own results then the submitter node will be notified. Also during the execution, the health of the partial jobs are followed using the heartbeats from the agents. If heartbeat message is not received from the agent within a certain period of time, the job cancellation is requested from the agent and the thread is migrated to another agent. If the agent can perform the task to a certain extent but not fully perhaps due to limited resources, then the process migration of the partial computation takes place by transferring the process state file to another available agent. Agents have the ability to migrate tasks among themselves and only the controller is informed and the status of the task in the controller is updated. Task reassignment capability of agents is a feature of autonomy. Also the information send to controller are used for monitoring. Although all these operations are carried out through a controller node, the status backup is done by another controller in the cluster.

A backup mechanism is triggered when these fail-overs happen; (i) the submitter detects that the primary controller is no longer sending heartbeats, and concludes that it had died. The submitter connects to the backup controller and requests a new state snapshot, (ii) the backup controller starts to receive snapshot requests from the submitter, and detects that the primary controller is down. The backup controller becomes primary, (iii) the backup server applies its pending list according to process state requests. We assume that at least one server will keep running. When the primary fails, the second controller will take over as long as more than one host is specified. When the primary returns to service, it notifies the backup. The backup then saves the state, the primary reads the saved state and takes overs the computation and the second controller returns the backup mode again.

Computation: The main component of computation is the compute (worker) node. A compute node is a virtual concept and represents the subcomponents (may be physical/virtual machines or software modules) where the main computing operations are shown in Figure 1, Computation block. Agent (basatd) is the foremost of these components that communicate with the Controller node. It is responsible for task scheduling, container creation, check-pointing. A job is divided into parts by the Controller and then converted into a task structure for the Agent. Task is ordered by the Agent. Before the task processing stage, the container in which the execution takes place must be prepared. An agent may use a scheduling mechanism to create containers statically or dynamically based on the nature of the task. For the container technology, we choose LXC (Linux containers) which relies on the Linux kernel cgroups with namespace isolation functionality. An agent may run different task scheduling algorithms. While the related task is running, the agent sends status information to the Controller. If the task completes successfully, a success report is sent to the Controller and the container is prepared for the next task. The commissioning and tracking of a new task is managed by the Agent task scheduler. If there are no tasks in the task queue, a new job-part is requested from the Controller. If the task fails during the execution, the Controller will be informed and another attempt will be made. If the Controller approves, the task is retried or the task is transferred to another agent. Check-pointing may be done with third party solutions (DMTCP, CRIU).

Data is stored on nodes provided by the distributed file system that is shown in Figure 1, Data Access block. BasatNet prefers Glusterfs over HDFS. Glusterfs is a decentralised-friendly distributed file system on the market today. The data layer consists of multiple data nodes that not only keeps the data distributed but also the log files and results. Using GlusterFS, the concept of data locality may be achieved with configuration such that the application has the least network latency. Data nodes may deliver data to worker nodes and controllers using HTTP, SSH, NFS like URL schemes, Libglusterfs API, and local mount. Data locality may be specified with an option in the configuration file at task scheduling.

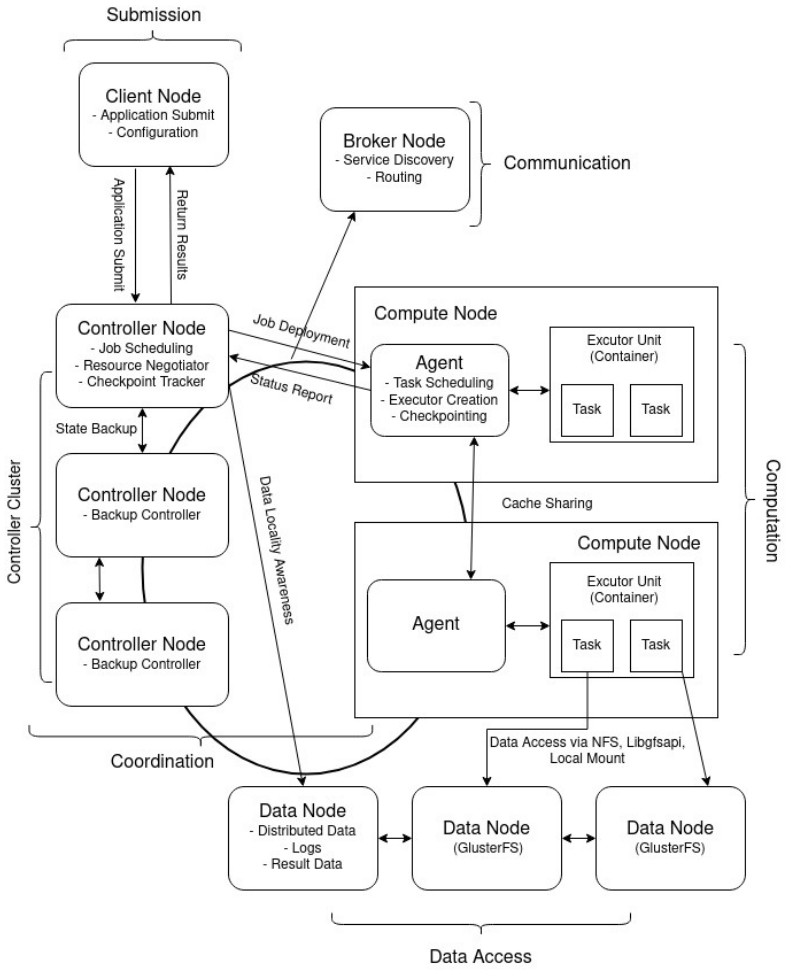

Figure 1. High level design of BasatNet.

Communication: The performance of distributed and parallel computing is directly affected by the design and implementation of the communication subsystem. We attempted to build a communication layer that not only utilizes the existing distributed messaging libraries such as MPI but also extends it with new patterns. ZeroMQ is used as the communication library in the proposed design. ZeroMQ behaves like an embedded networking library but also supports concurrency. The communication middleware established with ZeroMQ will manage the connections of modules that communicate with the Controller. ZeroMQ's built-in primitives for formal scalable communication is considered for BasatNet design. For the Controller's cluster the binary

star pattern was selected. In this pattern, the Controller backs up the coordination status to the next backup controller. The connection of the controller with the Agents is established using the router/dealer with a pub/subpattern. The communication is done through messages transmitted in binary form via ZeroMQ. The message types to be communicated include the heartbeat, task processing, job submit and data node metainformation. In addition to the implementation of the proposed communication protocol, BasatNet also supports NFS for data nodes, HTTP(S) for connections and MPI for parallel programming.

4. Proof-of-concept implementation

The C programming language was used for network related modules due to its low resource consumption and high performance. Lua was chosen to implement the settings in configuration subsystem due to its metalanguage features. Lua also has a very small virtual machine that is accessible from C. Python was used to test the system due to its ease of implementation with battery-included libraries. ZeroMQ was used for communication and data exchange over the network. Testing was conducted on Milis Linux operating system [1]. BasatNet has an open source code and available at the designated Git Repository [2]. Detailed implementation of components are reported in the following subsections.

4.1. Programmable configuration

Lua has a small virtual machine that may be embedded inside C by including header files. C programs may accesses the Lua environment in Stack to read configuration and exchange data. ProtoBuffer Lua implementation was used to transmit configurations with binary serialization as in Controller and other agents transmission. An example task configuration may be submitted with Lua as:

```
submit={type="cmd", content="python test.py", min_ram=32, MPI=yes,
Multi_Thread=no, Data_Parallel=true,}
```

4.2. Service discovery and router

This module was implemented as a broker as shown in Figure 1 *Communication block*, under the name of Basatnetd that handles the traffic between agents, controller and submitters. Basatnetd also checks the vitality of nodes through heartbeat such as Zookeeper [5]. Basatnetd does not handle key/value storage as in our design key/value operations are executed by the Controller. Basatnet broker is based on advanced ZeroMQ communication protocol that defines a reliable service-oriented request-reply dialog between a set of client applications, a broker and a set of worker applications.

4.3. Controller/job scheduler

The Controller orchestrates the system and together with an Agent performs the work as shown in Figure 1, Coordination block. While the Controller processes the job, the Agent processes the tasks of the job. The Controller performs these functions; (i) queuing submitted jobs, (ii) dividing jobs into related tasks according to the job configuration, (iii) distributing tasks to the agents, (iv) collecting the results from the agents, and (v) reporting the merged outcome of the tasks to the client as the result of the job. A fixed-size queue data structure is implemented as prototype. The scheduler function handles resource management and data/task

[1] Milis Linux (2021). Milis Linux Operating System [online]. Website https://mls.akdeniz.edu.tr [accessed 11 April 2022].
[2] BasatNet Source Code [online]. Website https://mls.akdeniz.edu.tr/git/milisarge/basatnet [accessed 11 April 2022].

parallelization. A custom heartbeat message sent by each agent to the controller plays a key role in resource management. This heartbeat message includes resource information (CPU, Ram, IO).

4.4. Agent/task scheduler

Agent is an component derived from workers like the Controller. Agents run on nodes that do the computation. Each task is queued by the Agent's own scheduler mechanism by utilizing the polling feature of ZeroMQ. In our implementation, a special file format is used to store task messages and process the FIFO queue. When a task is submitted to the Agent from the Controller, it is queued. Agents handle the task in a multithreaded environment while serving other tasks. Each task runs according to the cgroup policy of the Agent. After finishing the task, the results are sent to the Controller over the coordinator node.

4.5. Resource limitation

A process limitation is applied at each node for task execution using the *cgroups* feature of Linux kernel by specifying a policy for the tasks. Control groups and the subsystems that are related may be managed using shell commands and utilities. The Linux kernel provides different cgroup subsystems for CPU, RAM and Network. For each cgroup, the steps are; creating hierarchies, mounting of the necessary file systems, creation of cgroups, and setting of subsystem parameters (resource limits). An example configuration for BasatNet daemon is shown below:

```
group basatd_group{
  cpu    { cpu.shares = 100; }, memory { memory.limit_in_bytes = 256m; },
  blkio  { blkio.throttle.read_bps_device = "8:0 4194304";},
}
```

The example configuration above states that the cpu shares up-to 100 available memory of 256MB and disk read is limited to throughput for 4MiB/s. Read I/O to the primary disk shown by major:minor number is 8:0; that is 4MiB/s converted to bytes per second as $4 \times 1024 \times 1024 = 4{,}194{,}304$. Also we need to set the processes (tasks) for which it limits the related resources to the cgroups. This is done at cgrules.conf file under /etc directory. We defined our basatd_group inside the config file. At the beginning of line, there is basatd which shows the user in our behalf at Linux system. Each task is run by this user with the corresponding cgroups policy.

5. Results and discussion

As described in Section 4, a test environment based on isolated chroot system was implemented. This environment was provided with the help of Linux container technology, with Milis Linux system as the host system. A container image is deployed for each component included in Figure 1. Container images were obtained using Alpine Linux so that the container images were not too bloated. They are approximately 7-8 MB. For the test process, four containers were used respectively; Router (basatnet), Controller (basatctl), Worker (basatd) and Client (basatcli) nodes. The connection of the containers with each other was made via a virtual bridge network and the host network interface. Each component that is added to the test environment has a one-time deployment cost. Other samples were cloned.

The measurements given in Figure 2 were obtained by running BasatNet Router in a Linux Container. The memory was limited to 3MB to demonstrate that the system may run with a low memory footprint. With

this limitation, we determined that the maximum number of containers that can concurrently run were as follows: i) The container itself consumes up to 1.38 MB of memory. ii) When the application starts, the total memory consumption increases to 2.05MB in the listening mode. At this mode there is no worker node in the network. iii) Next, the controller starts as worker number 1. iv) Under the 3 MB limit, the system allows up to 3 more workers for actual computation. Consequently, for 3 Workers, 1 Controller and 1 Router, a net memory usage of 1.62MB (3MB–1.38MB) was observed.

When running a constant time job that sleeps at regular intervals in the testbed environment, we have achieved the following results:

- Fast deployment of controller and worker nodes using the Linux container structure for cloning. Only the cost of rootfs copying speed was required.
- Initiating client nodes may specify their requests with a flexible programmable configuration system. This allows unmet requests to be stored with its state in the container by taking a snapshot of it for future processing.
- Controller node freezes idle worker nodes and when a job arrives it may unfreeze the idle worker node in less than 1 s to join the distributed computing pool.
- Powerful communication layer enables communication over IPC (interprocess communication), TCP and UDP. Different distributed system patterns should be adaptable based on the needs of the job. Therefore, the communication library must be capable in handling heterogeneous protocols and allow easy implementation.

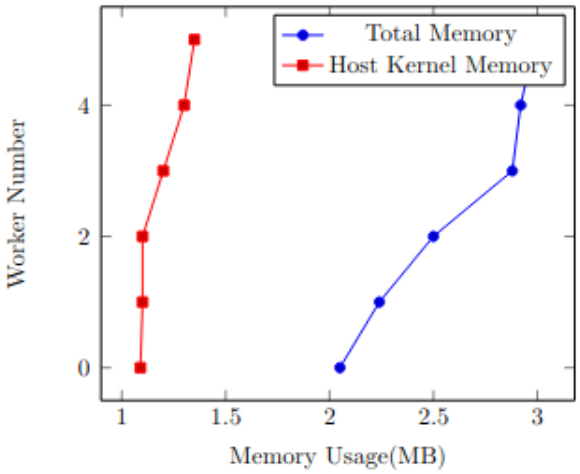

Figure 2. BasatNet Router Memory Usage $< 3MB$.

6. Conclusion and future work

Due to the ever-increasing data volume, the need for rapid data processing has the potential to directly affect scientific studies. For this reason, the convergence between HPC and big data continues inevitably, and investments in supercomputers are now being made in this field. Therefore, for a successful outcome, it is

important to develop the distributed computing environment correctly. However, this is difficult because, while Big Data assumes low cost commodity hardware for computation, HPC is designed to run on super computers.

For future work we envision that computation units may use Unikernel [29] like level container technology. Alternatively, using user-space programming, we can implement in kernel-space in favor of reducing context switch. Resource management and fault-tolerance skills will be developed by ensuring that the Coordinator has autonomous qualifications in decision making. Consequently, with this study, a distributed system design and implementation on a Linux host system that will work using different architectures has been presented. In the experimental environment we have implemented, the basic functioning of the components in the distributed system with isolated running environments with adjustable resources has been shown.

References

[1] Saraladevi B, Pazhaniraja N, Victer P, Basha S, Dhavachelvanc P. Big Data and Hadoop-a Study in Security Perspective. Procedia Computer Science 2015; 597-601. doi: 10.1016/j.procs.2015.04.091

[2] Gunay M, Ince M. Building an Open Source Big Data Platform Based on Milis Linux. Trends in Data Engineering Methods for Intelligent Systems 2021; 33-41. doi: 10.1007/978-3-030-79357-9_5

[3] Shvachko K, Kuang H, Radia S, Chansler R. The Hadoop Distributed File System. IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST) 2010; 1-10. doi: 10.1109/MSST.2010.5496972

[4] Kang S, Lee S, Lee K. Performance Comparison of OpenMP, MPI, and MapReduce in Practical Problems. Advances in Multimedia 2015;1-9. doi: 10.1155/2015/575687

[5] Hunt P, Konar M, Junque F, Reed B. ZooKeeper: Wait-Free Coordination for Internet-Scale Systems. Proceedings of the USENIX Conference on USENIX Annual Technical Conference 2010; 11. doi: 10.5555/1855840.1855851

[6] Dean J, Ghemawat G. MapReduce: Simplified Data Processing on Large Clusters. Communications of the ACM 2004;51: 137-150. doi: 10.1145/1327452.1327492

[7] Alexandrov A, Ewen S, Heimel M, Hueske F, Kao O et al. MapReduce and PACT - Comparing Data Parallel Programming Models. BTW 2011;180: 25-44. isbn: 978-3-88579-274-1

[8] Zheng Y, Kamil A, Driscoll M, Shan H, Yelick K. UPC++: A PGAS Extension for C++. IEEE 28th International Parallel and Distributed Processing Symposium 2014; 1105-1114. doi: 10.1109/IPDPS.2014.115

[9] Ayyalasomayajula H, West K. Experiences running different work load managers across Cray Platforms. InCoS2016 and Cray User Group 2017.

[10] Mohd U, Mengchen L, Min C. Job schedulers for Big data processing in Hadoop environment: Testing real-life schedule with benchmark programs. Digital Communications and Networks 2017; 1105-1114. doi: 10.1016/j.dcan.2017.07.008

[11] Yoo A, Jette M, Grondona M. SLURM: Simple Linux Utility for Resource Management. Job Scheduling Strategies for Parallel Processing 2003; 44-60. doi: 10.1007/10968987_3

[12] Ullah S, Awan D, Khiyal M. Big Data in Cloud Computing: A Resource Management Perspective. Scientific Programming 2018; 5418679:1-5418679:17. doi: 10.1155/2018/5418679

[13] Vavilapalli V, Murthy A, Dougkas C, Agarwal S, Konar M et al. Apache Hadoop YARN: Yet Another Resource Negotiator. Proceedings of the 4th Annual Symposium on Cloud Computing 2013; 5 (16). doi: 10.1145/2523616.2523633

[14] Hindman B, Konwinski A, Zaharia M, Ghodsi A, Joseph A et al. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation 2011; 14: 295-308. doi: 10.5555/1972457.1972488

[15] Rahman M, Lu X, Islam N, Rajachandrasekar R, Panda D. High-Performance Design of YARN MapReduce on Modern HPC Clusters with Lustre and RDMA. IEEE International Parallel and Distributed Processing Symposium 2015; p291-300. doi: 10.1109/IPDPS.2015.83

[16] Renker G, Stringfellow N, Howard K, Sadaf R, Trofinoff S. Deploying SLURM on XT , XE , and Future Cray Systems. Cray User Group 2011.

[17] Sudsee B, Kaewkasi C. An Improvement of a Checkpoint-based Distributed Testing Technique on a Big Data Environment. 21st International Conference on Advanced Communication Technology (ICACT) 2019; p1081-1090. doi: 10.23919/ICACT.2019.8702037

[18] Hargrove P, Duell J. Berkeley lab checkpoint/restart (BLCR) for Linux clusters. Journal of Physics: Conference Series 2006; 46: 494-499. doi: 10.1088/1742-6596/46/1/067

[19] Ansel J, Arya K, Cooperman G. DMTCP: Transparent checkpointing for cluster computations and the desktop. IEEE International Symposium on Parallel Distributed Processing 2009; 46: 1-12. doi: 10.1109/IPDPS.2009.5161063

[20] Pickartz S, Eiling N, Lankes S, Razik L, Monti A. Migrating LinuX Containers Using CRIU. ISC High Performance 2016; 674-684. doi:10.1007/978-3-319-46079-6_47

[21] Hintjens P. ZeroMQ: Messaging for Many Applications. 2013.

[22] Ahn D, Garlick J, Grondona M, Lipari D, Springmeyer B et al. Flux: A Next-Generation Resource Management Framework for Large HPC Centers. Proceedings of the 2014 43rd International Conference on Parallel Processing Workshops 2014; 9: 9-17. doi: 10.1109/ICPPW.2014.15

[23] Ahn D, Bass N, Chu A, Garlick J, Grondona M et al. Flux: Overcoming scheduling challenges for exascale workflows. Future Generation Computer Systems 2020; 110: 202-213. doi: 10.1016/j.future.2020.04.006

[24] Kamburugamuve S, Fox G. Designing Twister2: Efficient Programming Environment Toolkit for Big Data. 2017. doi: 10.13140/RG.2.2.18363.31524

[25] Kamburugamuve S, Kannan G, Wickramasinghe P, Abeykoon V, Fox G. Twister2: Design of a Big Data Toolkit. Concurrency and Computation: Practice and Experience 2019. doi: 10.1002/cpe.5189

[26] Wickramasinghe P, Kamburugamuve S, Govindarajan K, Abeykoon V, Widanage C. Twister2: TSet High-Performance Iterative Dataflow. International Conference on High Performance Big Data and Intelligent Systems (HPBD IS) 2019; p55-60. doi: 10.1109/HPBDIS.2019.8735495

[27] Fox G, Qiu J, Kamburugamuve S, Jha S, Luckow A. HPC-ABDS High Performance Computing Enhanced Apache Big Data Stack. 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing 2015; 1057-1066. doi: 10.1109/CCGrid.2015.122

[28] Ekanayake S, Kamburugamuve S, Fox G. SPIDAL Java: High Performance Data Analytics with Java and MPI on Large Multicore HPC Clusters. 24th High Performance Computing Symposium (HPC) 2016. doi: 10.13140/RG.2.1.3991.6568

[29] Madhavapeddy A, Scott J. Unikernels: Rise of the Virtual Library Operating System: What If All the Software Layers in a Virtual Appliance Were Compiled within the Same Safe, High-Level Language Framework?. Association for Computing Machinery 2013; (15): 30-44. doi: 10.1145/2557963.2566628

[30] Selvaganesan M, Liazudeen M. An Insight about GlusterFS and Its Enforcement Techniques. International Conference on Cloud Computing Research and Innovations (ICCCRI) 2016; 120-127. doi: 10.1109/ICCCRI.2016.26

[31] Saha P, Beltre A, Govindaraju M. Exploring the Fairness and Resource Distribution in an Apache Mesos Environment. IEEE 11th International Conference on Cloud Computing (CLOUD) 2018; 434-441. doi: 10.1109/CLOUD.2018.00061