Master node fault tolerance in distributed big data processing clusters

I. Gankevich Yu. Tipikin V. Korkhov
V. Gaiduchok A. Degtyarev A. Bogdanov

Saint Petersburg State University,
Dept. of Computer Modelling and Multiprocessor Systems
Universitetskaia emb. 7-9, 199034 Saint Petersburg, Russia
Email: i.gankevich@spbu.ru
Email: y.tipikin@spbu.ru
Email: v.korkhov@spbu.ru
Email: gvladimiru@gmail.com
Email: a.degtyarev@spbu.ru
Email: bogdanov@csa.ru

Abstract: Distributed computing clusters are often built with commodity hardware which leads to periodic failures of processing nodes due to relatively low reliability of such hardware. While worker node fault-tolerance is straightforward, fault tolerance of master node poses a bigger challenge. In this paper master node failure handling is based on the concept of master and worker roles that can be dynamically re-assigned to cluster nodes along with maintaining a backup of the master node state on one of worker nodes. In such case no special component is needed to monitor the health of the cluster while master node failures can be resolved except for the cases of simultaneous failure of master and backup. We present experimental evaluation of the technique implementation, show benchmarks demonstrating that a failure of a master does not affect running job, and a failure of a backup results in re-computation of only the last job step.

Keywords: parallel computing; Big Data processing; distributed computing; backup node; state transfer; delegation; cluster computing; fault-tolerance; high-availability; hierarchy

Biographical notes:
Ivan Gankevich is a research assistant in computer science at Dept. of Computer Modelling and Multiprocessor Systems, Saint Petersburg State University. His research interests include middleware for high-performance computing, parallel programming, and large-scale ocean waves simulations.

Yuri Tipikin is a PhD student in computer science at Dept. of Computer Modelling and Multiprocessor Systems, Saint Petersburg State University. His research interests include middleware for high-performance computing, task scheduling and resource allocation algorithms.

Dr. Vladimir Korkhov is an associate professor at the Computer Modeling and Multiprocessor Systems department, Faculty of Applied Mathematics and Control Processes, St. Petersburg State University, Russia. He received PhD degree from the University of Amsterdam in 2009 with the thesis on hierarchical resource management in Grid computing; he participated in a number of national and international projects on distributed and grid computing, and a number of RFBR-funded projects. As a post-doctoral researcher he worked at the Academic Medical Center of the University of Amsterdam and at Charité — Medical

University of Berlin on applying grid technology to bioinformatics and medical applications. Research interests include parallel, distributed, grid and cloud computing, resource management, workflows. Dr. Korkhov has published around 70 scientific papers in international journals and conference proceedings.

Vladimir Gaiduchok is a PhD student in computer science at Dept. of Computer Science and Engineering, Saint Petersburg Electrotechnical University "LETI". His research interests include parallel programming, cloud computing and optimisation.

Alexander Degtyarev graduated from Leningrad Shipbuilding Institute in 1985. Defended PhD thesis in 1991 (thesis in the field of computational fluid dynamics). Research positions in Institute for High Performance Computing and Data Bases and different universities. Teaching experience from 1993, professor in computational sciences from 2005. Current courses in the field of high performance computing, intelligence systems (in St. Petersburg State University). Areas of research: development of mathematical and computer models for complex dynamic systems, application of high performance computing technology, on-board intelligence systems. The author of more than 100 papers.

Main activities of Prof. Bogdanov are related to mathematical methods in physics, computational methods and mathematical modeling. Lately he became concentrated on creation of data and knowledge bases for a number of applied fields and creation of algorithms for high-performance computing. At the same time, he's involved with computational cluster creation problems on various computer platforms, i.e. system integration problems.

This paper is a revised and expanded version of a paper entitled "Factory: Master node high-availability for Big Data applications and beyond" presented at the 16th International Conference on Computational Science and its Applications (ICCSA'16), Beijing, China, July 4–7.

1 Introduction

Fault tolerance of data processing pipelines is one of the top concerns in development of job schedulers for big data processing, however, most schedulers provide fault tolerance for subordinate nodes only. These types of failures are routinely mitigated by restarting the failed job or its part on healthy nodes, and failure of a master node is often considered either improbable, or too complicated to handle and configure on the target platform. System administrators often find alternatives to application level fault tolerance: they isolate master node from the rest of the cluster by placing it on a dedicated machine, or use virtualisation technologies instead. All these alternatives complexify configuration and maintenance, and by decreasing probability of a machine failure resulting in a whole system failure, they increase probability of a human error.

From such point of view it seems more practical to implement master node fault tolerance at the application level, however, there is no generic implementation. Most implementations are too tied to a particular application to become universally acceptable. We believe that this happens due to people's habit to think of a cluster as a collection of individual machines each of which can be either master or slave, rather than to think of a cluster as a whole with master and slave roles being *dynamically* assigned to a particular physical machine.

This evolution in thinking allows to implement middleware that manages master and slave roles automatically and handles node failures in a generic way. This software provides an API to distribute parallel tasks on the pool of available nodes and among them. Using this API one can write an application that runs on a cluster without knowing the exact number of online nodes. The middleware is implemented as a daemon running on each cluster node which acts as an intermediate point of communication for distributed applications and transparently routes application messages between operating system processes running on different cluster nodes.

The rest of the paper is organised as follows. In Section 2 we discuss how dynamic node role assignment is implemented in state-of-the-art middleware systems and how hierarchical dependencies between kernels help establish rules for restarting failed parts of a parallel application. In Section 3 we describe programming model based on kernels — small independent units of work — and an algorithm of handling master node failure. In Section 4 and 5 we present and discuss results of experiments that show validity of the proposed approach and conclude the paper in Section 6.

The research for new fault-tolerance methods is motivated by the fact that the size of large-scale computing systems (clusters and supercomputers) approaches critical point, where the number of nodes is so large that probability of all nodes simultaneously working without a faulure tends to nought. In other words, in future large-scale systems it is highly probable that a parallel application experience node failure throughout its execution, and tolerating this failure in a transparent way and without checkpointts will increase performance of future parallel applications.

2 Related work

Dynamic role assignment is an emerging trend in design of distributed systems [26, 8, 5, 3, 21], however, it is still not used in big data job schedulers. For example, in popular YARN [29] and Spark [30] job schedulers, which are used by Hadoop and Spark big data analysis frameworks, master and slave roles are static. Failure of a slave node is tolerated by restarting a part of a job on a healthy node, and failure of a master node is tolerated by setting up standby reserved server [22]. Both master servers are coordinated by Zookeeper service which itself uses dynamic role assignment to ensure its fault-tolerance [25]. So, the whole setup is complicated due to Hadoop scheduler lacking dynamic roles: if dynamic roles were available, Zookeeper would be redundant in this setup. Moreover, this setup does not guarantee continuous operation of the master node because the standby server needs time to recover current state after a failure.

The same problem occurs in high-performance computing where the master node of a job scheduler is the single point of failure. In [27, 10] the authors use replication to make the master node highly-available, but the backup server role is assigned statically and cannot be delegated to a healthy worker node. This solution is closer to fully dynamic role assignment than the high-availability solution for big data schedulers, because it does not involve using external service to store configuration which should also be highly-available, however, it is far from ideal solution where roles are completely decoupled from physical servers.

Finally, the simplest master node high-availability is implemented in Virtual Router Redundancy Protocol (VRRP) [20, 18, 23]. Although VRRP protocol does provide master and backup node roles, which are dynamically assigned to available routers, this protocol works on top of the IPv4 and IPv6 protocols and is designed to be used by routers and

reverse proxy servers. Such servers lack the state that needs to be restored upon a failure (i.e. there is no job queue in web servers), so it is easier for them to provide high-availability. On Linux it is implemented in Keepalived routing daemon [6].

In contrast to web servers and HPC and Big Data job schedulers, some distributed key-value stores and parallel file systems have symmetric architecture, where master and slave roles are assigned dynamically, so that any node can act as a master when the current master node fails [26, 8, 5, 3, 21]. This design decision simplifies management and interaction with a distributed system. From system administrator point of view it is much simpler to install the same software stack on each node than to manually configure master and slave nodes. Additionally, it is much easier to bootstrap new nodes into the cluster and decommission old ones. From a user point of view, it is much simpler to provide web service high-availability and load-balancing when you have multiple backup nodes to connect to.

Dynamic role assignment would be beneficial for Big Data job schedulers because it allows to decouple distributed services from physical nodes, which is the first step to build highly-available distributed service. The reason that there is no general solution to this problem is that there is no generic programming environment to write and execute distributed programmes. The aim of this work is to propose such an environment and to describe its internal structure.

The programming model used in this work is partly based on the well-known actor model of concurrent computation [2, 17]. Our model borrows the concept of actor — an object that stores data and methods to process it; this object can react to external events by either changing its state or producing more actors. We call this objects *computational kernels*. Their distinct feature is hierarchical dependence on parent kernel that created each of them, which allows implementing fault-tolerance by restarting failed subordinate kernel.

However, using hierarchical dependence alone is not enough to develop high-availability of a master kernel — the first kernel in a parallel programme. To solve the problem the other part of our programming model is based on bulk-synchronous parallel model [28]. It borrows the concept of superstep — a sequential step of a parallel programme; at any time a programme executes only one superstep, which allows implementing high-availability of the first kernel (under assumption that it has only one subordinate at a time) by sending it along its subordinate to a different cluster node thus making a distributed copy of it. Since the first kernel has only one subordinate at a time, its copy is always consistent with the original kernel. This eliminates the need for complex distributed transactions and distributed consensus algorithms and guarantees protection from at most one master node failure per superstep.

To summarise, the framework developed in this paper protects a parallel programme from failure of any number of subordinate nodes and from one failure of a master node per superstep. The paper does not answer the question of how to determine if a node has failed, it assumes node failure when the network connection to a node is prematurely closed. In general, the presented research goes in line with further development of the virtual supercomputer concept coined and evaluated in [12, 13, 4].

3 Methods

3.1 Model of computation

To infer fault tolerance model which is suitable for big data applications we use bulk-synchronous parallel model [28] as the basis. This model assumes that a parallel programme is composed of a number of sequential steps that are internally parallel, and global synchronisation of all parallel processes occurs after each step. In our model all sequential steps are pipelined where it is possible. The evolution of the computational model is described as follows.

Given a programme that is sequential and large enough to be decomposed into a number of sequential steps, the simplest way to make it run faster is to exploit data parallelism. Usually it means finding multi-dimensional arrays and loops that access their elements and trying to make them parallel. After transforming the loops the programme will still have the same number of sequential steps, but every step will (ideally) be internally parallel.

After that the only possibility to speedup the programme is to overlap execution of code blocks that work with different hardware devices. The most common pattern is to overlap computation with network or disk I/O. This approach makes sense because all devices operate with little synchronisation, and issuing commands in parallel makes the whole programme perform better. This behaviour can be achieved by allocating a separate task queue for each device and submitting tasks to these queues asynchronously with execution of the main thread. After this optimisation the programme will be composed of a number of steps chained into the pipeline, each step is implemented as a task queue for a particular device.

Pipelining of otherwise sequential steps is beneficial not only for the code accessing different devices, but for the code different branches of which are suitable for execution by multiple hardware threads of the same core, i.e. branches accessing different regions of memory or performing mixed arithmetic (floating point and integer). In other words, code branches which use different modules of processor are good candidates to run in parallel on a processor core with multiple hardware threads.

Even though pipelining may not add parallelism for a programme that uses only one input file (or a set of input parameters), it adds parallelism when the programme can process multiple input files: each input generates tasks which travel through the whole pipeline in parallel with tasks generated by other inputs. With a pipeline an array of files is processed in parallel by the same set of resources allocated for a batch job. The pipeline is likely to deliver greater efficiency for busy HPC clusters compared to executing a separate job for each input file, because the time that each subsequent job spends in the queue is eliminated. A diagram of a pipeline is presented in fig. 1 and discussed in sec 4.

Computational model with a pipeline can be seen as *bulk-asynchronous model*, because of the parallel nature of otherwise sequential execution steps. This model is the basis of the fault-tolerance model developed here.

3.2 Programming model principles

Data processing pipeline model is based on the following principles that maximise efficiency of a programme:

- There is no notion of a message in the model, a kernel is itself a message that can be sent over network to another node and directly access any kernel on the local node. Only programme logic may guarantee the existence of the kernel.

- A kernel is a *cooperative routine*, which is submitted to the kernel pool upon the call and is executed asynchronously by the scheduler. There can be any number of calls to other subroutines inside routine body. Every call submits corresponding subroutine to the kernel pool and returns immediately. Kernels in the pool can be executed in any order; this fact is used by the scheduler to exploit parallelism offered by the computer by distributing kernels from the pool across available cluster nodes and processor cores.

- Asynchronous execution prevents the use of explicit synchronisation after the call to subroutine is made; the system scheduler returns the control flow to the routine each time one of its subroutine returns. Such cooperation transforms each routine which calls subroutines into an event handler, where each event is a subroutine and the handler is the routine that called them.

- The routine may communicate with any number of local kernels, whose addresses it knows; communication with kernels which are not adjacent in the call stack complexifies the control flow and the call stack looses its tree shape. Only the programme logic may guarantee the presence of communicating kernels in memory. One way to ensure this is to perform communication between subroutines which are called from the same routine. Since such communication is possible within the hierarchy through the parent routine, it may be treated as an optimisation that eliminates the overhead of transferring data over an intermediate node. The situation is different for interactive or event-based programmes (e.g. servers and programmes with graphical interface) in which this is primary type of communication.

- In addition to this, communication which does not occur along hierarchical links and is executed over the cluster network complexifies the design of resiliency algorithms. Since it is impossible to ensure that a kernel resides in memory of a neighbour node, a node may fail in the middle of its execution of the corresponding routine. As a result, upon a failure of a routine all of its subroutines must be restarted. This encourages a programmer to construct

  – deep tree hierarchies of tightly coupled kernels (which communicate on the same level of hierarchy) to reduce overhead of recomputation;
  – fat tree hierarchies of loosely coupled kernels, providing maximal degree of parallelism.

  Deep hierarchy is not only the requirement of technology; it helps optimise communication of large number of cluster nodes reducing it to communication of adjacent nodes.

Thus, kernels possess properties of both cooperative routines and event handlers.

3.3 Fail over model

Although fault-tolerance and high-availability are different terms, in essence they describe the same property — an ability of a system to switch processing from a failed component

to its live spare or backup component. In case of fault-tolerance it is the ability to switch from a failed slave node to a spare one, i.e. to repeat computation step on a healthy slave node. In case of high-availability it is the ability to switch from a failed master node to a backup node with full recovery of execution state. These are the core abilities that constitute distributed system's ability to *fail over*.

The key feature that is missing in the current parallel programming and big data processing technologies is a possibility to specify hierarchical (parent-child) dependencies between parallel tasks (kernels). When one has such dependency, it is trivial to determine which kernel should be responsible for re-executing a failed kernel on a healthy node. To re-execute the kernel on the top of the hierarchy, its copy is created and executed on a different node (detailed algorithm is discussed in sec. 3.5). There exists a number of engines that are capable of executing directed acyclic graphs of kernels in parallel [1, 19], but graphs are not suitable to infer parent-child relationship between kernels, because a node in the graph may have multiple parent nodes.

3.4 Programming model

This work is based on the results of previous research: In [16, 15] we developed an algorithm that allows to build a tree hierarchy from strictly ordered set of cluster nodes. The sole purpose of this hierarchy is to make a cluster more fault-tolerant by introducing multiple master nodes. If a master node fails, then its subordinates try to connect to another node from the same or higher level of the hierarchy. If there is no such node, one of the subordinates becomes the master. In [14] we developed a framework for big data processing without fault tolerance, and here this framework is combined with fault-tolerance techniques described in this paper.

Each programme that runs on top of the tree hierarchy is composed of computational kernels—objects that contain data and code to process it. To exploit parallelism a kernel may create arbitrary number of subordinate kernels which are automatically spread first across available processor cores, second across subordinate nodes in the tree hierarchy. The programme is itself a kernel (without a parent as it is executed by a user), which either solves the problem sequentially on its own or creates subordinate kernels to solve it in parallel.

In contrast to HPC applications, in big data applications it is inefficient to run computational kernels on arbitrary chosen nodes. More practical approach is to bind every kernel to a file location in a parallel file system and transfer the kernel to that location before processing the file. That way expensive data transfer is eliminated, and the file is always read from a local drive. This approach is more deterministic compared to existing ones, e.g. MapReduce framework runs jobs on nodes that are "close" to the file location, but not necessarily the exact node where the file is located [7]. However, this approach does not come without disadvantages: scalability of a big data application is limited by the strategy that was employed to distribute its input files across cluster nodes. The more nodes used to store input files, the more read performance is achieved. The advantage of our approach is that the I/O performance is more predictable, than one of the hybrid approach with streaming files over the network.

The main purpose of the model is to simplify the development of distributed batch processing applications and middleware. The focus is to make an application resilient to failures, i.e. make it fault tolerant and highly available, and do it transparently to a programmer. The implementation is divided into two layers: the lower layer consists of routines and classes for single node applications (with no network interactions), and the

upper layer for applications that run on an arbitrary number of nodes. There are two kinds of tightly coupled entities in the model — *control flow objects* (or *kernels*) and *pipelines* — which are used together to compose a programme.

Kernels implement control flow logic in theirs `act` and `react` methods and store the state of the current control flow branch. Both logic and state are implemented by a programmer. In the `act` method some function is either directly computed or decomposed into nested functions (represented by a set of subordinate kernels) which are subsequently sent to a pipeline. In `react` method subordinate kernels that returned from the pipeline are processed by their parent. Calls to `act` and `react` methods are asynchronous and are made within threads attached to a pipeline. For each kernel `act` is called only once, and for multiple kernels the calls are done in parallel to each other, whereas `react` method is called once for each subordinate kernel, and all the calls are made in the same thread to prevent race conditions (for different parent kernels different threads may be used).

Pipelines implement asynchronous calls to `act` and `react`, and try to make as many parallel calls as possible considering concurrency of the platform (no. of cores per node and no. of nodes in a cluster). A pipeline consists of a kernel pool, which contains all the subordinate kernels sent by their parents, and a thread pool that processes kernels in accordance with rules outlined in the previous paragraph. A separate pipeline is used for each device: There are pipelines for parallel processing, schedule-based processing (periodic and delayed tasks), and a proxy pipeline for processing of kernels on other cluster nodes.

In principle, kernels and pipelines machinery reflect the one of procedures and call stacks, with the advantage that kernel methods are called asynchronously and in parallel to each other (as much as programme logic allows). Kernel field is the stack, `act` method is a sequence of processor instructions before nested procedure call, and `react` method is a sequence of processor instructions after the call. Constructing and sending subordinate kernels to the pipeline is nested procedure call. Two methods are necessary to make calls asynchronous, and replace active wait for completion of subordinate kernels with passive one. Pipelines, in turn, allow implementing passive wait, and call correct kernel methods by analysing their internal state.

3.5 Handling master node failures

A possible way of handling a failure of a node where the first kernel is located (a master node) is to replicate this kernel to a backup node, and make all updates to its state propagate to the backup node by means of a distributed transaction. This approach requires synchronisation between all nodes that execute subordinates of the first kernel and the node with the first kernel itself. When a node with the first kernel goes offline, the nodes with subordinate kernels must know what node is the backup one. However, if the backup node also goes offline in the middle of execution of some subordinate kernel, then it is impossible for this kernel to discover the next backup node to return to, because this kernel has not discovered the unavailability of the master node yet. One can think of a consensus-based algorithm to ensure that subordinate kernels always know where the backup node is, but distributed consensus algorithms do not scale well to the large number of nodes and they are not reliable [11]. So, consensus-based approach does not play well with asynchronous nature of computational kernels as it may inhibit scalability of a parallel programme.

Fortunately, the first kernel usually does not perform operations in parallel, it is rather sequentially launches execution steps one by one, so it has only one subordinate at a time. Such behaviour is described by bulk-synchronous parallel programming model, in the

Table 1 NDBC dataset properties.

Dataset size	144MB
Dataset size (uncompressed)	770MB
No. of wave stations	24
Time span	3 years (2010–2012)
Total no. of spectra	445422

framework of which a programme consists of sequential supersteps which are internally parallel [28]. Keeping this in mind, we can simplify synchronisation of its state: we can send the first kernel along with its subordinate to the subordinate node. When the node with the first kernel fails, its copy receives its subordinate, and no execution time is lost. When the node with its copy fails, its subordinate is rescheduled on some other node, and in the worst case a whole step of computation is lost.

The described approach works only for kernels that do not have a parent and have only one subordinate at a time, and act similar to manually triggered checkpoints. The advantage is that they

- save results after each sequential step when memory footprint of a programme is low,
- they save only relevant data,
- and they use memory of a subordinate node instead of stable storage.

4 Evaluation

Master node fail over technique is evaluated on the example of wave energy spectra processing application. This programme uses NDBC dataset [24] to reconstruct frequency-directional spectra from wave rider buoy measurements and compute variance. Each spectrum is reconstructed from five variables using the following formula [9].

$$S(\omega,\theta) = \frac{1}{\pi}\left[\frac{1}{2} + r_1\cos\left(\theta-\alpha_1\right) + r_2\sin\left(2\left(\theta-\alpha_2\right)\right)\right] S_0(\omega).$$

Here ω denotes frequency, θ is wave direction, $r_{1,2}$ and $\alpha_{1,2}$ are parameters of spectrum decomposition and S_0 is non-directional spectrum; $r_{1,2}$, $\alpha_{1,2}$ and S_0 are acquired through measurements. Properties of the dataset which is used in evaluation are listed in Table 1.

The algorithm of processing spectra is as follows. First, current directory is recursively scanned for input files. Data for all buoys is distributed across cluster nodes and each buoy's data processing is distributed across processor cores of a node. Processing begins with joining corresponding measurements for each spectrum variables into a tuple, then for each tuple frequency-directional spectrum is reconstructed and its variance is computed. Results are gradually copied back to the machine where the application was executed and when the processing is complete the programme terminates. A data processing pipeline corresponding to the algorithm is presented in fig. 1.

In a series of test runs we benchmarked performance of the application in the presence of different types of failures:

- failure of a master node (a node where the first kernel is run),

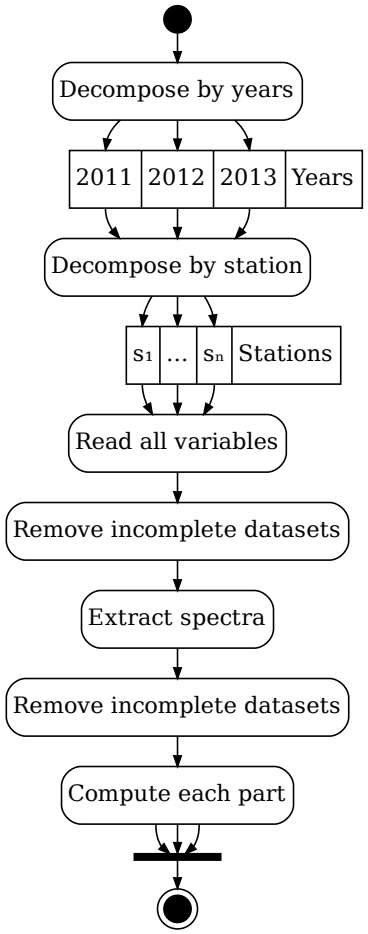

Figure 1 Data processing pipeline of a programme.

Table 2 Test platform configuration.

CPU	Intel Xeon E5440, 2.83GHz
RAM	4Gb
HDD	ST3250310NS, 7200rpm
No. of nodes	12
No. of CPU cores per node	8

Table 3 Benchmark parameters.

Experiment no.	Master node	Victim node	Time to offline, s
1	root		
2	root	leaf	30
3	leaf	leaf	30
4	leaf	root	30

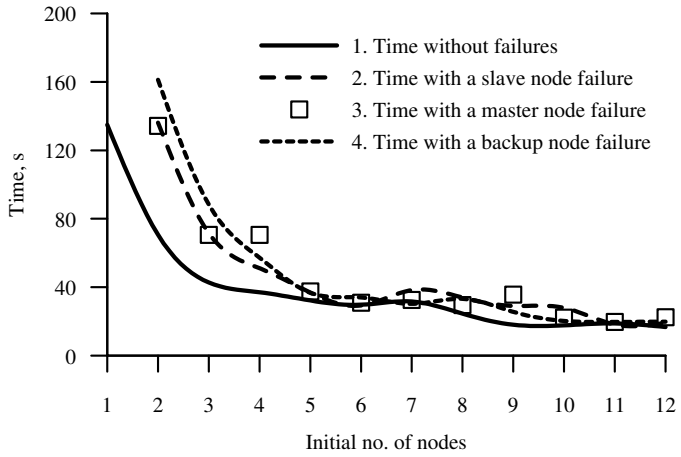

Figure 2 Performance of spectrum processing application in the presence of different types of node failures.

- failure of a slave node (a node where spectra from a particular station are reconstructed) and
- failure of a backup node (a node where the first kernel is copied).

A tree hierarchy with sufficiently large fan-out value was chosen to make all cluster nodes connect directly to the first one so that only one master node exists in the cluster. In each run the first kernel was launched on a different node to make mapping of kernel hierarchy to the tree hierarchy optimal. A victim node was made offline after a fixed amount of time early after the programme start. To make up for the node failure all data files have replicas stored on different cluster nodes. All relevant parameters are summarised in Table 3 (here "root" and "leaf" refer to a node in the tree hierarchy). The results of these runs were compared to the run without node failures (Figure 2).

The benchmark showed that only a backup node failure results in significant performance penalty, in all other cases the performance is roughly equals to the one without failures but with the number of nodes minus one. It happens because the backup node not only stores the copy of the state of the current computation step but executes this step in parallel with other subordinate nodes. So, when a backup node fails, the master node executes the whole step once again on arbitrarily chosen healthy subordinate node.

To measure how much time is lost due to a failure we divide the total execution time with a failure by the total execution time without the failure but with the number of nodes minus one. The results for this calculation are obtained from the same benchmark and are presented in fig. 3. The difference in performance lies within 20% margin for the number of

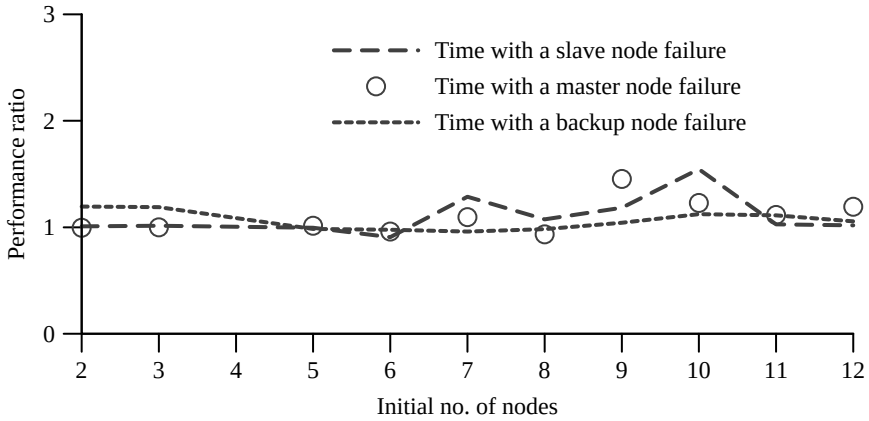

Figure 3 Slowdown of the spectrum processing application in the presence of different types of node failures compared to execution without failures but with the number of nodes minus one.

nodes less or equal to six. As is expected, the slowdown is noticable for the small number of nodes, and performance penalty descreases for large number of nodes, as smaller portion of execution state is lost. The picture is different for the number of nodes larger than six: performance ratio fluctuates within 50% margin. The reason for such behaviour is that spectrum processing application does not scale beyond six nodes, and the benchmark does not give realiable results beyond this point.

5 Discussion

Described algorithm guarantees to handle one failure per computational step, more failures can be tolerated if they do not affect the master node. The system handles simultaneous failure of all subordinate nodes, however, if both master and backup nodes fail, there is no chance for an application to survive. In this case the state of the current computation step is lost, and the only way to restore it is to restart the application.

Computational kernels are means of abstraction that decouple a distributed application from physical hardware: it does not matter how many nodes are online for an application to run successfully. Computational kernels eliminate the need to allocate a physical backup node to make the master node highly available, with computational kernels approach any node can act as a backup one. Finally, computational kernels can handle subordinate node failures in a way that is transparent to a programmer.

The disadvantage of this approach is evident: there is no way of making existing middleware highly available without rewriting their source code. Although, our programming framework is lightweight, it is not easy to map architecture of existing middleware systems to it: most systems are developed keeping in mind static assignment of server/client roles, which is not easy to make dynamic. Nevertheless, our approach may simplify design of future middleware systems.

6 Conclusion

Dynamic roles assignment is beneficial for Big Data applications and distributed systems in general. It decouples architecture of a distributed system from underlying hardware as much as possible, providing highly-available service on top of varying number of physical machines. As much as virtualisation simplifies management and administration of a computer cluster, our approach may simplify development of reliable distributed applications which run on top of the cluster.

Acknowledgements

The research was carried out using computational resources of Resource Centre "Computational Centre of Saint Petersburg State University" (T-EDGE96 HPC-0011828-001) within frameworks of grants of Russian Foundation for Basic Research (projects no. 16-07-01111, 16-07-00886, 16-07-01113).

References

[1] Bilge Acun, Arpan Gupta, Nikhil Jain, Akhil Langer, Harshitha Menon, Eric Mikida, Xiang Ni, Michael Robson, Yanhua Sun, Ehsan Totoni, et al. Parallel programming with migratable objects: Charm++ in practice. In *High Performance Computing, Networking, Storage and Analysis, SC14: International Conference for*, pages 647–658. IEEE, 2014.

[2] Gul A Agha. Actors: A model of concurrent computation in distributed systems. Technical report, DTIC Document, 1985.

[3] J Chris Anderson, Jan Lehnardt, and Noah Slater. *CouchDB: The definitive guide*. O'Reilly Media, Inc., Sebastopol, CA, 2010.

[4] A. Bogdanov, A. Degtyarev, V. Korkhov, V. Gaiduchok, and I. Gankevich. *Virtual Supercomputer as basis of Scientific Computing*, volume 11, chapter 5, pages 159–198. Nova Science Publishers, 2015.

[5] Eric B Boyer, Matthew C Broomfield, and Terrell A Perrotti. Glusterfs one storage server to rule them all. Technical report, Los Alamos National Laboratory (LANL), 2012.

[6] Alexandre Cassen. Keepalived: Health checking for LVS & high availability. *URL http://www.linuxvirtualserver.org*, 2002. Accessed: 2016-03-11.

[7] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[8] Manda Sai Divya and Shiv Kumar Goyal. Elasticsearch: An advanced and quick search technique to handle voluminous data. *Compusoft*, 2(6):171, 2013.

[9] Marshall D Earle. Nondirectional and directional wave data analysis procedures. Technical report, NDBC, 1996.

[10] Christian Engelmann, Stephen L Scott, Chokchai Box Leangsuksun, Xubin Ben He, et al. Symmetric active/active high availability for high-performance computing system services. *Journal of Computers*, 1(8):43–54, 2006.

[11] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[12] I. Gankevich, V. Gaiduchok, D. Gushchanskiy, Y. Tipikin, V. Korkhov, A. Degtyarev, A. Bogdanov, and V. Zolotarev. Virtual private supercomputer: Design and evaluation. In *CSIT 2013 - 9th International Conference on Computer Science and Information Technologies, Revised Selected Papers*, pages 1–6, 2013.

[13] I. Gankevich, V. Korkhov, S. Balyan, V. Gaiduchok, D. Gushchanskiy, Y. Tipikin, A. Degtyarev, and A. Bogdanov. Constructing virtual private supercomputer using virtualization and cloud technologies. In *Computational Science and Its Applications - ICCSA 2014, Lecture Notes in Computer Science*, volume 8584, pages 341–354, 2014.

[14] Ivan Gankevich and Alexander Degtyarev. Efficient processing and classification of wave energy spectrum data with a distributed pipeline. *Computer Research and Modeling*, 7(3):517–520, 2015.

[15] Ivan Gankevich, Yuri Tipikin, Alexander Degtyarev, and Vladimir Korkhov. Novel approaches for distributing workload on commodity computer systems. In *Computational Science and Its Applications - ICCSA 2015, Lecture Notes in Computer Science*, volume 9158, pages 259–271, 2015.

[16] Ivan Gankevich, Yuri Tipikin, and Vladimir Gaiduchok. Subordination: Cluster management without distributed consensus. In *International Conference on High Performance Computing & Simulation (HPCS)*, pages 639–642. IEEE, 2015.

[17] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 235–245. Morgan Kaufmann Publishers Inc., 1973.

[18] R Hinden et al. Virtual router redundancy protocol (vrrp); rfc3768. txt. *IETF Standard, Internet Engineering Task Force, IETF, CH*, pages 0000–0003, 2004.

[19] Mohammad Islam, Angelo K Huang, Mohamed Battisha, Michelle Chiang, Santhosh Srinivasan, Craig Peters, Andreas Neumann, and Alejandro Abdelnur. Oozie: towards a scalable workflow management system for Hadoop. In *Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies*, page 4. ACM, 2012.

[20] S Knight, D Weaver, D Whipple, R Hinden, D Mitzel, P Hunt, P Higginson, M Shand, and A Lindem. Rfc2338. *Virtual Router Redundancy Protocol*, 1998.

[21] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[22] Arun C Murthy, Chris Douglas, Mahadev Konar, Owen O'Malley, Sanjay Radia, Sharad Agarwal, and Vinod KV. Architecture of next generation

apache hadoop mapreduce framework. URL `https://issues.apache.org/jira/secure/attachment/12486023/MapReduce_NextGen_Architecture.pdf`, 2011. Accessed: 2016-03-11.

[23] S Nadas. Rfc 5798: Virtual router redundancy protocol (vrrp) version 3 for ipv4 and ipv6. *Internet Engineering Task Force (IETF)*, 2010.

[24] NDBC directional wave stations. URL: `http://www.ndbc.noaa.gov/dwa.shtml`. Accessed: 2016-03-11.

[25] Ekpe Okorafor and Mensah Kwabena Patrick. Availability of jobtracker machine in hadoop/mapreduce zookeeper coordinated clusters. *Advanced Computing: An International Journal (ACIJ)*, 3(3):19–30, 2012.

[26] David Ostrovsky, Yaniv Rodenski, and Mohammed Haji. *Pro Couchbase Server*. Apress, New York, 2015.

[27] Kai Uhlemann, Christian Engelmann, and Stephen L Scott. Joshua: Symmetric active/active replication for highly available hpc job and resource management. In *Cluster Computing, 2006 IEEE International Conference on*, pages 1–10. IEEE, 2006.

[28] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[29] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM, 2013.

[30] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, page 14. USENIX Association, 2012.