# Factory:
# Master node high-availability for Big Data applications and beyond

Ivan Gankevich, Yuri Tipikin, Vladimir Korkhov, Vladimir Gaiduchok, Alexander Degtyarev, and Alexander Bogdanov

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

**Abstract.** Master node fault-tolerance is the topic that is often dimmed in the discussion of big data processing technologies. Although failure of a master node can take down the whole data processing pipeline, this is considered either improbable or too difficult to encounter. The aim of the studies reported here is to propose rather simple technique to deal with master-node failures. This technique is based on temporary delegation of master role to one of the slave nodes and transferring updated state back to the master when one step of computation is complete. That way the state is duplicated and computation can proceed to the next step regardless of a failure of a delegate or the master (but not both). We run benchmarks to show that a failure of a master is almost "invisible" to other nodes, and failure of a delegate results in recomputation of only one step of data processing pipeline. We believe that the technique can be used not only in Big Data processing but in other types of applications.

**Keywords:** parallel computing · Big Data processing · distributed computing · backup node · state transfer · delegation · cluster computing · fault-tolerance

## 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 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 works as a cluster operating system overlay allowing to write distributed applications.

## 2 Related work

Dynamic role assignment is an emerging trend in design of distributed systems [3, 5, 8, 21, 26], however, it is still not used in big data job schedulers. For example, in popular YARN job scheduler [29], which is 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 master node because standby server needs time to recover current state after a failure.

The same problem occurs in high-performance computing where master node of a job scheduler is the single point of failure. In [10, 27] the authors use replication to make the master node highly-available, but 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 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) [18, 20, 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. In 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 [3,5,8,21,26]. 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 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 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 to implement fault-tolerance based on simple restart of a 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 to implement 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 failed, it assumes a 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 [4, 12, 13].

## 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 several 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 several 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 several 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 I/O 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. So, after this optimisation, the programme will be composed of several 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 code accessing different devices, but for 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, and possibly with greater efficiency for busy HPC clusters compared to executing a separate job for each input file, because the time that each subsequent job after the first spends in a queue is eliminated.

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 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 restoration 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 dependencies between parallel tasks. When one has such dependency, it is trivial to determine which task should be responsible for re-executing a failed task on a healthy node. To re-execute the root of the hierarchy, a backup root task is created and executed on a different node. There exists a number of engines that are capable of executing directed acyclic graphs of tasks in parallel [1, 19], but graphs are not good to infer master-slave relationship between tasks, because a node in the graph may have multiple parent nodes.

## 3.3 Programming model

This work is based on the results of previous research: In [15, 16] 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 hybrid approach with streaming files over the network.

### 3.4 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 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.

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 Results

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 $\omega$ denotes frequency, $\theta$ 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.

**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 |

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 application was executed and when the processing is complete the programme terminates.

**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 |

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),
– 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 1).

**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 |

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 a 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.

## 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 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 master node highly-available, with computational kernels approach any node can act as a backup one. Finally,

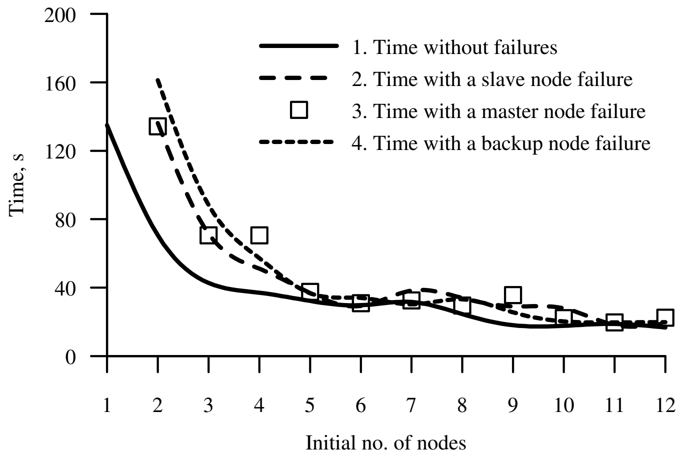

**Fig. 1.** Performance of spectrum processing application in the presence of different types of node failures.

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. Hopefully, our approach will 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) and Saint Petersburg State University (project no. 0.37.155.2014).

## References

1. Acun, B., Gupta, A., Jain, N., Langer, A., Menon, H., Mikida, E., Ni, X., Robson, M., Sun, Y., Totoni, E., et al.: Parallel programming with migratable objects: Charm++ in practice. In: High Performance Computing, Networking, Storage and Analysis, SC14: International Conference for. pp. 647–658. IEEE (2014)
2. Agha, G.A.: Actors: A model of concurrent computation in distributed systems. Tech. rep., DTIC Document (1985)
3. Anderson, J.C., Lehnardt, J., Slater, N.: CouchDB: The definitive guide. O'Reilly Media, Inc. (2010)
4. Bogdanov, A., Degtyarev, A., Korkhov, V., Gaiduchok, V., Gankevich, I.: Virtual supercomputer as basis of scientific computing. in series: Horizons in Computer Science Research v. 11, pp. 159–198 (2015)
5. Boyer, E.B., Broomfield, M.C., Perrotti, T.A.: Glusterfs one storage server to rule them all. Tech. rep., Los Alamos National Laboratory (LANL) (2012)
6. Cassen, A.: Keepalived: Health checking for lvs & high availability. URL http://www.linuxvirtualserver.org (2002)
7. Dean, J., Ghemawat, S.: MapReduce: Simplified data processing on large clusters. Communications of the ACM 51(1), 107–113 (2008)
8. Divya, M.S., Goyal, S.K.: Elasticsearch: An advanced and quick search technique to handle voluminous data. Compusoft 2(6), 171 (2013)
9. Earle, M.D.: Nondirectional and directional wave data analysis procedures. Tech. rep., NDBC (1996)
10. Engelmann, C., Scott, S.L., Leangsuksun, C.B., He, X.B., et al.: Symmetric active/active high availability for high-performance computing system services. Journal of Computers 1(8), 43–54 (2006)
11. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. Journal of the ACM (JACM) 32(2), 374–382 (1985)
12. Gankevich, I., Gaiduchok, V., Gushchanskiy, D., Tipikin, Y., Korkhov, V., Degtyarev, A., Bogdanov, A., Zolotarev, V.: Virtual private supercomputer: Design and evaluation. In: CSIT 2013 - 9th International Conference on Computer Science and Information Technologies, Revised Selected Papers. pp. 1–6 (2013)
13. Gankevich, I., Korkhov, V., Balyan, S., Gaiduchok, V., Gushchanskiy, D., Tipikin, Y., Degtyarev, A., Bogdanov, A.: Constructing virtual private supercomputer using virtualization and cloud technologies. In: Computational Science and Its Applications - ICCSA 2014, Lecture Notes in Computer Science. vol. 8584, pp. 341–354 (2014)
14. Gankevich, I., Degtyarev, A.: Efficient processing and classification of wave energy spectrum data with a distributed pipeline. Computer Research and Modeling 7(3), 517–520 (2015), `http://crm-en.ics.org.ru/journal/article/2301/`
15. Gankevich, I., Tipikin, Y., Degtyarev, A., Korkhov, V.: Novel approaches for distributing workload on commodity computer systems. In: Computational Science and Its Applications - ICCSA 2015, Lecture Notes in Computer Science. vol. 9158, pp. 259–271 (2015)
16. Gankevich, I., Tipikin, Y., Gaiduchok, V.: Subordination: Cluster management without distributed consensus. In: International Conference on High Performance Computing & Simulation (HPCS). pp. 639–642. IEEE (2015)
17. Hewitt, C., Bishop, P., Steiger, R.: A universal modular actor formalism for artificial intelligence. In: Proceedings of the 3rd international joint conference on Artificial intelligence. pp. 235–245. Morgan Kaufmann Publishers Inc. (1973)

18. Hinden, R., et al.: Virtual router redundancy protocol (vrrp); rfc3768. txt. IETF Standard, Internet Engineering Task Force, IETF, CH pp. 0000–0003 (2004)
19. Islam, M., Huang, A.K., Battisha, M., Chiang, M., Srinivasan, S., Peters, C., Neumann, A., Abdelnur, A.: Oozie: towards a scalable workflow management system for Hadoop. In: Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies. p. 4. ACM (2012)
20. Knight, S., Weaver, D., Whipple, D., Hinden, R., Mitzel, D., Hunt, P., Higginson, P., Shand, M., Lindem, A.: Rfc2338. Virtual Router Redundancy Protocol (1998)
21. Lakshman, A., Malik, P.: Cassandra: A decentralized structured storage system. ACM SIGOPS Operating Systems Review 44(2), 35–40 (2010)
22. Murthy, A.C., Douglas, C., Konar, M., O'Malley, O., Radia, S., Agarwal, S., KV, V.: Architecture of next generation apache hadoop mapreduce framework. Apache Jira (2011)
23. Nadas, S.: 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`
25. Okorafor, E., Patrick, M.K.: Availability of jobtracker machine in hadoop/mapreduce zookeeper coordinated clusters. Advanced Computing: An International Journal (ACIJ) 3(3), 19–30 (2012)
26. Ostrovsky, D., Rodenski, Y., Haji, M.: Pro Couchbase Server. Apress (2015)
27. Uhlemann, K., Engelmann, C., Scott, S.L.: Joshua: Symmetric active/active replication for highly available hpc job and resource management. In: Cluster Computing, 2006 IEEE International Conference on. pp. 1–10. IEEE (2006)
28. Valiant, L.G.: A bridging model for parallel computation. Communications of the ACM 33(8), 103–111 (1990)
29. Vavilapalli, V.K., Murthy, A.C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., et al.: Apache hadoop yarn: Yet another resource negotiator. In: Proceedings of the 4th annual Symposium on Cloud Computing. p. 5. ACM (2013)