Novel approaches for distributing workload on commodity computer systems

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

Saint Petersburg State University,
Universitetskii 35, Petergof, 198504, Saint Petersburg, Russia
igankevich@yandex.com, yuriitipikin@gmail.com

Abstract. Efficient management of a distributed system is a common problem for university's and commercial computer centres, and handling node failures is a major aspect of it. Failures which are rare in a small commodity cluster, at large scale become common, and there should be a way to overcome them without restarting all parallel processes of an application. The efficiency of existing methods can be improved by forming a hierarchy of distributed processes. That way only lower levels of the hierarchy need to be restarted in case of a leaf node failure, and only root node needs special treatment. Process hierarchy changes in real time and the workload is dynamically rebalanced across online nodes. This approach makes it possible to implement efficient partial restart of a parallel application, and transactional behaviour for computer centre service tasks.

Keywords: long-lived transactions, distributed pipeline, node discovery, software engineering, distributed computing, cluster computing

Introduction

There are two main tasks for a computer centre: to run users' parallel applications, and to service users. Often these tasks are delegated to some distributed systems, so that users can service themselves, and often these systems are heterogeneous and there are many of them in a computer centre. Yet another system is introduced to make them consistent, but this system is usually not distributed. Moreover, the system usually does not allow rolling back a distributed transaction spanning its subordinate systems. So, the use of reliable distributed systems under control of an unreliable one makes the whole system poorly orchestrated, and absence of automatic error-recovery mechanism increases amount of manual work to bring the system up after a failure.

To orchestrate computations and perform service tasks in distributed environment the system should be decomposed into two levels. The top level of this system is occupied by transaction manager which executes long-lived transactions (a transaction consisting of nested subordinate ones which spans a long period of time and provides relaxed consistency guarantees). Transactions are distributed across cluster nodes and are retried or rolled back in case of a system failure. On the second level a distributed pipeline is formed from cluster nodes to process compute-intensive workloads with automatic restart of failed processes. The goal of this decomposition is to separate service tasks which need transactional behaviour from parallel applications which need only restart of failed processes (without a roll back). On both levels of the system a hierarchy is used to achieve these goals.

The first level of the system is capable of reliably executing ancillary and configuration tasks which are typical to university's computer centres. Long-lived transactions allow for a task to run days, months and years, until the transaction is complete. For example, a typical task of registering a user in a computer centre for a fixed period of time (the time span of his/her research grant) starts after the user submits registration form and ends when the research is complete. Additional tasks (e.g. allocation of computational resources, changing quotas and custom configuration) are executed as subordinate of the main one. Upon completion of the work tasks are executed in reverse, reconfiguring the system to its initial state and erasing or archiving old data.

The rest of the paper describes the structure of two levels of the system and investigates their performance in a number of tests. Long-lived transactions are discussed in Section 1 and distributed pipeline (lower level) in Section 2.

1 Long-lived transactions

While performing computing in HPC environment various errors may occur. Hardware errors have the greatest impact among others. To fix this type of errors several approaches exist today which often consist of restarting the job completely. In distributed systems computational nodes have even more risk to be lost because of additional factors such as unreliable network. Thus, a complete job restart every time one or more nodes fail is ineffective.

While searching solution of this problem let us refer to the traditional transaction mechanism. Designed for operations on data it uses logging and locking to prevent data corruption and loss. ACID properties — Atomicity, Consistency, Isolation, Durability — of the transaction apply to relational databases, but for distributed system they are implemented with several restrictions. Now let us take a look at high-performance distributed environment. Here operations are performed on the tasks and the objective is to get valid computation results. At first glance, to apply transactions for computing one needs to segment initial task code and take a subtask as an atomic operation, but this is not sufficient.

Unlike database operations, computations can take much more time to execute, and long-lived transactions, which theoretically can take as much time as needed, is the solution. The main aspect of this technology is a correct logging and further journal processing. There is no unified definition of "long-lived transaction", so we define this term here.

Long-lived transaction is a transaction operations of which are executed for long time periods and there are long gaps between completion of operations. So, it is not safe to assume fast execution time of such transactions. For them only atomicity and reliability properties are guaranteed.

At the modeling stage web service can be a perfect container for computational subtask. Using REST [1] — representational state transfer — as a specific implementation of web services, we design and implement job scheduling on nodes as a call of a web service with target URL. Main accent here is on restore process of failed operation. The aim of this paper is to offer time-efficient algorithm of such restoration. Next, we will compare properties of REST realisation to "reliable" set of properties ACID and will draw a conclusion about meaningful changes in our model, which guarantee satisfiability of the initial task.

During testing REST web services inside transaction container, the fact of inapplicability of ACID "as is" was revealed. These properties conflict with both REST basics and a definition of a web service. Lets consider these properties step by step.

1.1 ACID in REST

Atomicity. Atomicity guarantees that transactions will not be partially saved and in terms of data this works best. However, web services are operations which create, modify and delete data while running, so transaction involving one or more web services must be moved to a higher level of abstraction. In fact, REST web service transactional system has two levels of atomicity: database level and level where web services are called directly. First of them is provided by a particular database implementation by default, second one is on the logical level, which programmer must implement in terms of a particular algorithm. It is important to understand what web service implementation must guarantee logical atomicity of its internal operations by providing only two available states of termination: absolutely correct result of entire web service and error state result. Thus, a collection of web services on logical level can be considered as a single web service recursively applying such requirements.

So, there is a need for a rollback operation for a web service, and in [2, 3] the authors also came to this conclusion. However, REST architecture has no mandatory requirements to system functionality implementation, and this generally leads to impossibility of automatic operation rollback in those systems. Even if rollback operations were implemented by web service developer, there would be no guarantee of valid result after calling these methods, because logical side of an algorithm can be non-trivial.

Durability. Durability is an interesting property. It prevents losing state information (even from hardware failure) by logging all actions in special journal. There are some challenges to implement an efficient distributed journal, but for now there are a few related articles where logging REST web services was partially described [2, 3]. In our approach to achieve efficient failed-task restoration distributed logging system plays an important role, and implementation of this property meets REST requirements.

Isolation. In REST this property is difficult to achieve without making an additional proxy server objects. Those objects serve as queues filtering “transactional” web services and executing them sequentially. This method is described in more detail in [2]. But if web service is designed to use parallel operations, a proxy server can be a performance bottleneck. In this case, isolation must be implemented on web service level, not on abstract level of a collection of web services. Transaction isolation through web service isolation imposes additional requirements to web service developer, but efficiency of transactional system may suffer without it. Isolation of a collection of transactions is applied by transactional job scheduler by default, because it executes transactions sequentially from a queue.

Consistency. Much like isolation, consistency is difficult to guarantee on web service level. It is a property of database rather than a web service.

1.2 Implementation

Main feature of our transactional manager is a special initial task code structure for long-lived transactions. Implementation consist of a server which executes transactions sequentially by placing them to a queue, logging and collecting responses, and a rewritten client code, which uses special functions (we called it *act* and *react*) to divide a task into a set of subtasks. On server’s input we have structured code of a subtask, transferred in JSON format, for example. Each subtask is an autonomous part of code. Initial task after rewriting by this algorithm is represented by a tree, which itself is a transaction and leaves are operations. Thus, sequence of operations are saved: sequential operations have parent-child relation, and parallel have child-child relation.

This approach is largely different from those described in [2, 3], it is not focused only on web service calls. In real world applications, reliable summary result is what user wants, without middleware web services calls information. But those middleware operations must be processed in any case. Only simplistic algorithms use exclusively web service calls, often a call is a result of subtask processing from another web service. Dividing task to a group of subtasks and after that formatting a transaction allows processing not only invokes of web services. In fact, any part of initial task can be secure.

In REST web services there is no universal way to achieve general atomicity. Rollback is implemented by moving the tree backward from a leave with failed state. Rollback performs for each subtask separately, not affecting other subtask on that particular computational node. In case of a node hardware failure, first rollback will wait the node to come online for some time, to not to move a task to another node’s queue. In case of impossibility of that scenario, rollback function goes one level up and tries the same approach. Code segmentation can improve restoration time significantly, but also prevent legacy application to run in such environment.

The ineffectiveness of running rollback straight away on higher levels (entire subtree) shown in [1]. As previously mentioned, rollback function is empty by default and must be written by a programmer of a web service himself. This step is a necessary and logical, because correct rollback for each function must be written by a person who exactly knows in which state abstract transaction will be after failed operation and what that operation was doing before stop. Transactional system described in this paper is a tool, not an universal solution for all types of operations because each computational algorithm is unique. Use of this tool requires rethinking of original algorithm in terms of partitioning to autonomous segments.

1.3 Building the transaction and early results

There is a step-by-step algorithm of transaction manager.

1. Programmer select self-sufficient parts of original algorithm – marking it as transaction operations.
2. Programmer writes a rollback function for all of such parts.
3. Structured code is sent to transaction server.
4. Server executes transaction by placing a root node (which includes a whole algorithm) to queue and then starts moving down across the code tree.
5. If processing is done without failures, iteration reaches leaves and starts going back by transferring successful results from children to their parents.
6. If processing is done with failures, transaction will run a rollback function on failed nodes and try to prevent massive rollback by slowly moving up the tree by one level at time.

The system was tested on 3 level algorithm tree which produces 25 lines in journal until it is completed. Measured time indicates how long transaction manager works to complete the task if a rollback function was called at specific log line (Fig. 1). Computation time without any rollback calls shown as dotted line. Time peaks in Fig. 1 belong to lines in journal that designate “execution” step of one or more subtasks. Total overhead oscillate from 5% to 15% because of fast prototype implementation on Node.js technology. The task itself consists of simple integration. Web services as task was tested as well, to investigate properties of long-lived transactions that are described above.

2 Distributed pipeline

The main idea of distributed pipeline is to create virtual topology of processes running on different computer cluster nodes and update it in real-time as infrastructure changes. The changes include nodes going offline and online, joining or leaving the cluster, replacement of any hardware component or an entire network node (including switches and routers), and other changes affecting system performance. Each change results in virtual topology being updated for a new infrastructure configuration.

The main purpose of distributed pipeline is to optimise performance of distributed low-level service tasks running on a computer cluster. Typically these

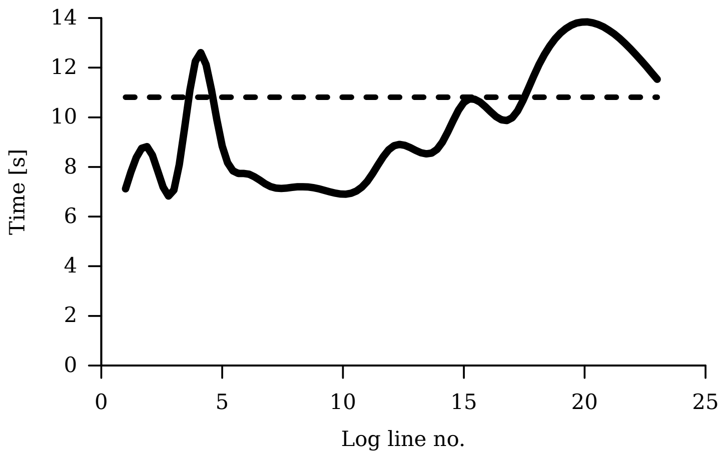

Fig. 1. Rollback time of a subtask at different log lines.

tasks involve querying each cluster node for some information or updating files on each node, and often single master node sends and collects messages from all slave nodes. To reduce network congestion intermediate nodes should be used to communicate master and slave nodes, that is to collect information and send it to master or the other way round. In case of large cluster a distributed pipeline with virtual topology of a tree is formed.

The other purpose of distributed pipeline is to improve efficiency of high-performance applications. These applications typically launch many parallel processes on a subset of cluster nodes, which communicate messages with each other. To make these communications efficient virtual topology should resemble a real one as much as possible and take into account nodes' performance (relative to each other) and communication link speed (see Section 2.3). That is, if two nodes are adjacent to each other in virtual topology, then the node which is closest to the tree root should have higher performance than the other one, and the link between them should be of the highest throughput.

Another aspect of high-performance applications is their fault tolerance and recovery time from node failure. Often these applications consist of long-running processes which are checkpointed with specified time period. If a node fails, then a new one is reserved and the application is recovered from the last checkpoint on each node. To reduce recovery time checkpointing can be made incremental and selective, that is to recover only those parts of a program that have failed and checkpoint the data that have actually changed. In distributed pipeline it is accomplished by logging messages sent to other nodes and resending them in case of a node failure (see Section 2.1).

To summarise, distributed pipeline creates virtual topology of a computer cluster in a form of a tree with high-performance nodes being closest to the root, and every virtual link having the highest-possible throughput. The purpose of this pipeline is to optimise performance of various cluster management tasks, but it can be beneficial for high-performance applications as well.

2.1 Implementation

From technical point of view distributed pipeline is a collection of reliable network connections between principal and subordinate nodes, which are made unique and shared by multiple applications (or service tasks) in a controlled way. The absence of duplicate connections between any two nodes conserves operating system resources and makes it possible to build distributed pipeline crossing multiple NAT environments: If the node hidden behind NAT can reach a node with public Internet address, they can communicate in both directions. Additionally, it allows creating persistent connections which are closed only when a change in topology occurs or in an event of system failure.

Connection between nodes can be shared by multiple applications in a controlled way. Each message is tagged with an application identifier (a number), and each application sends/receives messages from either standard input/output or a separate file descriptor (a pipe) which can be polled and operated asynchronously. The data is automatically converted to either portable binary (with network byte order) or text format. So, if high performance of asynchronous communication and small size of binary messages are not required, any programming language which can read and write data to standard streams can be used to develop an application for distributed pipeline.

To simplify writing high-performance applications for distributed pipeline the notion of a message is removed from the framework, instead the communication is done by sending one object to another and passing it as an argument to a defined method call. Each object can create subordinates to process data in parallel, perform parallel computations, or run tasks concurrently, and then report results to their principals. The absence of messages simplifies the API: an object always processes either principal's local data or results collected from subordinate objects.

Various aspects of reliable asynchronous communication have to be considered to make distributed pipeline fault-tolerant. If the communication between objects is not needed, the object is sent to a free node to perform some computation, and sent back to its principal when it is done. Objects which are sent to remote computer nodes are saved in a buffer and are removed from it when they return to their principals. That way even if the remote node fails after receiving the objects, they are sent back to their principals from the buffer with an error. After that it is principal that decides to rollback and resend objects to another node, or to fail and report to its own principal. In case the failure occurs before sending an object (e.g. node goes offline), then the object is sent to some other node bypassing its principal. To summarise, subordinate objects *always* return to their principals either with a success or a failure which further simplifies writing high-performance applications.

Principal/subordinate objects easily map to tree topology. If not specified explicitly, a principal sends a subordinate object to a local execution queue. In case of high load, it is extracted from the queue and sent to a remote node. If the remote node is also under high load, the object is sent further. So, the hierarchy of principals and subordinates is compactly mapped to tree topology: lower-level nodes are used on-demand when a higher-level node can not handle the flow of objects. In other words, when a higher-level node "overflows", the excessive objects are "spilled" to lower-level nodes.

So, the main goal of the implementation is to simplify application programming by minimising dependencies for small service programmes, making asynchronous messaging reliable, and by using automatic on-demand load balancing.

2.2 Peer discovery

The core of distributed pipeline is an algorithm which builds and updates virtual topology, and there are several states in which a node can be. In *disconnected* state a node is not a part of distributed pipeline, and to become the part it discovers its peers and connects to one of them. Prior to connecting the link speed and relative performance of a node are measured. After connecting the node enters *connected* state. In this state it receives updates from subordinate nodes (if any) and sends them to its principal. So, updates propagate from leaves to the root of a tree.

In initial state a node uses discovery algorithm to find its peers. The node queries operating system for a list of networks it is part of and filters out global and Internet host loopback addresses (/32 and 127.0.0.0/8 blocks in IPv4 standard). Then it sends a subordinate object to each address in the list and measures response time. In case of success the response time is saved in the table and in case of failure it is deleted. After receiving all subordinate objects principal repeats the process until minimal number of performance samples is collected for all peers. Then the rating (see Section 2.3) of each peer is calculated and the table is sorted by it. The principal declares the first peer in the table a leader and sends a subordinate object to it which increases peer's level by one. Then the whole algorithm repeats for the next level of a tree.

Sometimes two nodes in disconnected state can be chosen to be principals of each other, which creates a cycle in tree topology of distributed pipeline. This can happen if two nodes have the same rating. The cycle is eliminated by rejecting offer from the node with higher IP address, so that a node with lower IP address becomes the principal. To make rating conflicts rare IP address is used as the second field when sorting the table of peers.

On initial installation the total number of subordinate objects sent by each node amounts to mn^2 where n is the total number of nodes and m is the minimal number of samples, however, for subsequent restarts of the whole cluster this number can be significantly reduced with help of peer caches (Section 2.7). Upon entering connected state or receiving updates from peers each node stores current peer table in a file. When the node restarts it runs discovery algorithm only for peers in the file. If no peers are found to be online, then the algorithm repeats with an empty cache.

2.3 Node's rating

Determining performance of a computer is a complex task on its own right. The same computers may show varying performance for different applications, and the same application may show varying performance for different computers. For distributed pipeline performance of a node equals the number of computed objects per unit of time which depends on a type of a workload. So, performance of a computer is rather a function of a workload, not a variable that can be measured for a node.

In contrast to performance, concurrency (the ability to handle many workloads of the same type or a large workload) is a variable of a node often amounting to the number of processor cores. Using concurrency instead of processor speed for measuring performance is equivalent to assuming that all processors in a cluster have the same clock rate so that a number of cores determines performance. This assumption gives rough estimate of real performance, however, it allows determining performance just by counting the total number of cores in a computer node and relieves one from running resource-consuming performance tests on each node.

In [4–6] the authors suggest generalisation of Amdahl's law formula for computer clusters from which node's rating can be devised. The formula

$$S_N = \frac{N}{1 - \alpha + \alpha N + \beta\gamma N^3}$$

shows speedup of a parallel programme on a cluster taking into account communication overhead. Here N is the number of nodes, α is the sequential portion of a program, β is the diameter of a system (the maximal number of intermediate nodes a message passes through when any two nodes communicate), and γ is the ratio of node performance to network link performance. Speedup reaches its maximal value at $N = \sqrt[3]{(1-\alpha)/(2\beta\gamma)}$, so the higher the link performance is the higher speedup is achieved. In distributed pipeline performance is measured as the number of objects processed by a node or transmitted by network link during a fixed time period. The ratio of these two numbers is used as a rating.

So, when nodes are in disconnected state the rating is estimated as the ratio of a node concurrency to the response time. The rating of a remote node is re-estimated to be the ratio of number of transmitted objects to the number of processed objects per unit of time when nodes enter connected state and start processing objects.

2.4 Rating and level updates

A node in connected state may update its level if a new subordinate node with the same or higher level chooses it as a leader. The level of each node equals to the maximal level of subordinates plus one. So, if some high-level node connects to a principal, then its level is recalculated and this change propagates towards root of a tree. That way the root node level equals the maximal level of all nodes in the cluster plus one.

The rating of principal can become smaller than the rating of one of its subordinates when the workload type is changed from compute-intensive to data-intensive or the other way round. This also may occur due to a delayed level update, a change in node's configuration, or as a result of higher level node being offline. When it happens, the principal and the subordinate swap their positions in virtual topology, that is the higher level subordinate node becomes principal. Thus high-performance node can make its way to the root of a tree, if there are no network bottlenecks in the path.

So, rating and level updates propagate from leaves to the root of a tree in virtual topology automatically adapting distributed pipeline for a new type of workload or new cluster configuration.

2.5 Node failures and network partitions

There are three types of node failures in distributed pipeline: a failure of a leaf node, a principal node, and the root node. When a leaf node fails, objects in the corresponding sent buffer of its principal node are returned to their principals, and objects in unsent buffer are sent to some other subordinate node. If error processing is not done by principal object, then returning subordinate objects are also re-sent to other subordinate nodes. In case of principal or root node failure the recovery mechanism is the same, but subordinate nodes that lost their principal enter disconnected state, and a new principal is chosen from these subordinate nodes.

In case of root node failure all objects which were sent to subordinate nodes are lost and retransmitted one more time. Sometimes this results in restart of the whole application which discards previously computed objects. The obvious solution to this problem is to buffer subordinate objects returning to their principals so that in an event of a failure retransmit them to a new principal node. However, in case of a root node failure there is no way to recover objects residing in this particular node other than replicating them to some other node prior to failure. This makes the solution unnecessary complicated, so for now no simple solution to this problem has been found.

In case of network partition the recovery mechanism is also the same, and it also possess disadvantages of a root node failure: The results computed by subordinate objects are discarded and potentially large part of the application has to be restarted.

So, the main approach for dealing with failures consists of resending lost objects to healthy nodes which is equivalent of recomputing a part of the problem being solved. In case of root node failure or network partition a potentially large number of objects are recomputed, but no simple solution to this problem has been found.

2.6 Evaluation

Test platform consisted of a multi-processor node, and Linux network namespaces were used to consolidate virtual cluster of varying number of nodes on a physical node [7–9]. Distributed pipeline needs one daemon process on each node to operate correctly, so one virtual node was used for each daemon. Tests were repeated multiple times to reduce influences of processes running in background. Each subsequent test run was separated from previous one with a delay to give operating system time to release resources, cleanup and flush buffers.

Discovery test was designed to measure effect of cache on the time an initial node discovery takes. For the first run all cache files were removed from file system so that a node went from disconnected to connected state. For the second run all caches for each node were generated and stored in file system so that each node started from connected state.

Buffer test shows how many objects sent buffer holds under various load. Objects were sent between two nodes in a “ping-pong” manner. Every update of buffer size was captured and maximum value for each run was calculated. A delay between sending objects simulated the load on the system: the higher the load is the higher the delay between subsequent transfers is.

2.7 Test results

Discovery test showed that caching node peers can speed up transition from disconnected to connected state by up to 25 times for 32 nodes (Fig. 2), and this number increases with the number of nodes. This is expected behaviour since the whole discovery step is omitted and cached values are used to reconstruct peers’ level and performance data. The absolute time of this test is expected to increase when executed on real network, and cache effect might become more dramatic.

Buffer test showed that sent buffer contains many objects only under low load, i.e. when the ratio of computations to data transfers is low. Under high load computations and data transfer overlap, and the number of objects in sent buffer lowers (Fig. 3).

Related work

In [10] the authors discuss the use of message logging to implement efficient recovery from a failure. They observed that some messages written to log are commutative (can be reordered without changing the output of a program) and used this assumption to optimise recovery process. In our system objects in sent buffer are used instead of messages in a log, they are commutative within bounds of the buffer but do not represent history of all messages sent to another node. They are deleted upon receiving a reply, and deleted object can be safely ignored when recovering from a hard fault. So, when an object depends on parallel execution of its subordinates, results can be collected in any order, and it is often desirable to log only execution of principal to reduce recovery time.

In [11] the author introduces general distributed wave algorithms for leader election. In contrast to distributed pipeline, these algorithms assume that there can be only one leader and as a result do not take into account link speed

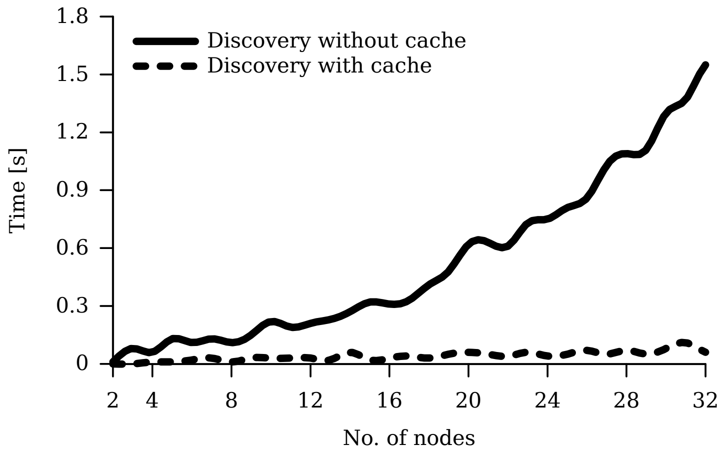

Fig. 2. Time taken for various number of nodes to discover each other with and without help of cache.

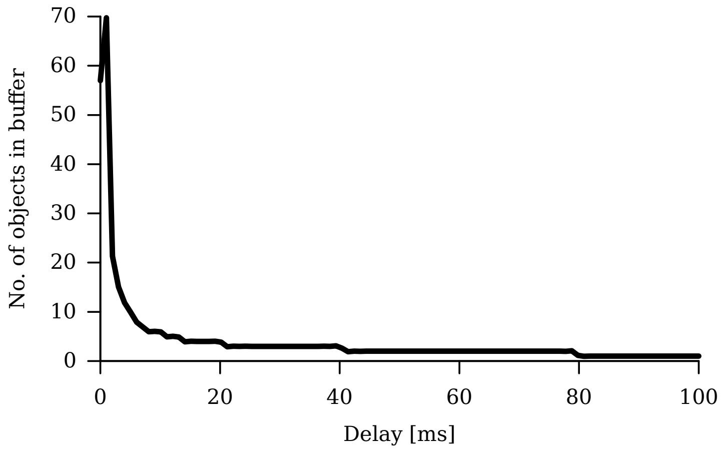

Fig. 3. Number of objects in sent buffer for various delays between objects transfers.

between nodes. The goal of discovery algorithm is to create a framework of leaders to distribute workload on a cluster, and each leader is found by recursively repeating election process for the new level of a hierarchy. Using hierarchy of leaders simplifies election algorithm, and taking into account links between nodes allows efficient mapping of resulting hierarchy to physical topology of a network.

Conclusions and future work

Distributed pipeline is a general method for distributing workload on a commodity cluster which relies on dynamically reconfigurable virtual topology and reliable data transfer. It primarily focuses on service tasks but can be used for high-performance computing as well. The future work is to find a simple way to make distributed pipeline resilient to the root node failures, and test applicability of this approach to high-performance computing applications.

Acknowledgement. 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 (project no. 13-07-00747) and Saint Petersburg State University (projects no. 9.38.674.2013 and 0.37.155.2014).

References

1. J. Armstrong, *Making reliable distributed systems in the presence of software errors.* PhD thesis, The Royal Institute of Technology Stockholm, Sweden, 2003.
2. S. Kochman, P. T. Wojciechowski, and M. Kmieciak, “Batched transactions for RESTful web services,” in *Current Trends in Web Engineering*, pp. 86–98, Springer, 2012.
3. E. Wilde and C. Pautasso, *REST: from research to practice.* Springer Science & Business Media, 2011.
4. A. Degtyarev, “High performance computer technologies in shipbuilding,” in *OPTIMISTIC — optimization in marine design, Mensch & Buch Verlag, Berlin* (L. Birk and S. Harries, eds.).
5. I. Soshmina and A. Bogdanov, “Using GRID technologies for computations (in russian),” *Saint Petersburg State University Bulletin (Physics and Chemistry)*, vol. 3, pp. 130–137, 2007.
6. S. Andrianov and A. Degtyarev, *Parallel and distributed computations (in Russian).* Saint Petersburg State University, 2007.
7. B. Lantz, B. Heller, and N. McKeown, “A network in a laptop: rapid prototyping for software-defined networks,” in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, p. 19, ACM, 2010.
8. N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown, “Reproducible network experiments using container-based emulation,” in *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, pp. 253–264, ACM, 2012.

9. B. Heller, *Reproducible Network Research with High-fidelity Emulation.* PhD thesis, Stanford University, 2013.
10. J. Lifflander, E. Meneses, H. Menon, P. Miller, S. Krishnamoorthy, and L. V. Kalé, "Scalable replay with partial-order dependencies for message-logging fault tolerance," in *IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 19–28, IEEE, 2014.
11. G. Tel, *Introduction to distributed algorithms.* Cambridge university press, 2000.