Achieving High Availability in D-Bobox

Miroslav Cermak, Filip Zavoral
Faculty of Mathematics and Physics
Charles University in Prague, Czech Republic
{cermak, zavoral}@ksi.mff.cuni.cz

***Abstract*—Using a distributed environment for data stream processing brings many challenges, especially when requiring an exact result from processing of big data. A distributed system is more vulnerable to failures as hardware crashes, software errors, or network malfunctions. Loss of node current state and loss of intermediate results due to node failure results in the restart of the computation, which increases the time of the computation and its cost and this is therefore unacceptable. Achieving high availability (HA) of such system brings some challenges. In this paper, we introduce our framework for parallel and distributed processing, D-Bobox, and its requirements on high availability implementation. We also describe the main high availability methods used today and discuss their applicability in our framework. Finally, we propose a solution how to obtain high availability in D-Bobox.**

Keywords–high availability; D-Bobox; stream computing; distributed computing;

I. INTRODUCTION

A new class of applications - stream processing systems (SPS) - is given a lot of attention in the last years [1], [2], [3]. These applications have to process high amount of low latency data streams, i.e., financial data processing, patients monitoring using various sensors, traffic analysis, etc. Stream processing seems to be effective not only in processing continuous data streams, but also in processing big static data as for example semantic databases [4]. Distributing sources and processing of big data at multiple nodes allows better scaling of computation performance in terms of data size. Stream processing system that uses advantages of the distributed environment are called distributed SPS (DSPS).

A typical approach to increase the performance of a distributed system is adding more computational nodes. However, this also increases the risk of a failure, which has a negative effect on the performance and dependability of a distributed system. Faults introduce errors into computation so we get wrong or incomplete results. However, many applications require that the system provides exact and same results each time running on the same input data. One of such systems can be a database system that also requires performance effectiveness and good performance/value ratio. Therefore, the presence of a high availability (HA) unit that handles recovery fast and correctly with minimal impact on failure-free processing is necessary for DSPSs.

To achieve HA in distributed stream processing systems, following tasks must be addressed:

1) periodic and incremental backup (or replication) of computing node state
2) error detection
3) choosing a failover node
4) lost state recovery after failure
5) manage network partition

In this work, we deal with the recovery from a node failure, so we pay our attention mainly to tasks (1), (3) and (4) as they have the biggest impact on the behavior and characteristic of each of the HA methods. Since error detection is mostly independent from the actual HA (recovery) method and the management of network splitting and partitioning is a specific type of failure concerning multiple nodes, we do not address these issues in this paper.

The paper is structured as follows: in Section II, we define recovery types according to [5]. In Section III we present contemporary HA methods that are discussed on the selected HA problems. The D-Bobox system is introduced in Section IV; in Section V we propose solutions for the integration of HA into D-Bobox.

II. RECOVERY TYPES

The ability to mask failure so it cannot be observed from final data stream is considered the fundamental requirement on HA algorithms. Consider a node U, that contains a set of n input data streams (I_1, .., I_n) and produce the output stream O. Computation e consists of processes like consuming, processing and producing data tuples. The data stream O_e is the result of the computation e at the node U. According to their handling of the O_f+O'=O equality, where O_f is computation result before failure, O' is output after recovery and O is failure free computation output, recovery types can be named as *gap recovery*, *rollback recovery* and *precise recovery*.

1) Gap recovery: is the simplest and least demanding recovery type. It manages node replacement after node failure detection. However, input and output preservation is not guaranteed and data loss is expected. Its main advantage is fast recovery time and almost no slowdown of failure free execution.

2) Rollback recovery: ensures that no information is lost during failure. According to operators used, the rollback recovery can be further divided into following subtypes:

- *repeatable* when exactly the same tuples are generated during recovery.
- *convergent* when different tuples from the same data are generated, and these tuples converge to the original tuples.
- *divergent* when different tuples from the same data are generated, and these tuples never converge to the

 ISBN: 978-1-61208-334-6

TABLE I. OUTPUTS PRODUCED BY EACH TYPE OF RECOVERY

Recovery type	Before failure			After failure				
Precise	t_1	t_2	t_3	t_4	t_5	t_6	t_7	...
Gap recovery	t_1	t_2	t_3			t_6	t_7	...
Rollback								
- repeatable	t_1	t_2	t_3	t_2	t_3	t_4	t_5	...
- convergent	t_1	t_2	t_3	t'_4	t'_5	t_6	t_7	...
- divergent	t_1	t_2	t_3	t'_4	t'_5	t'_6	t'_7	...

original ones. This is typical for non-deterministic operators.

3) Precise recovery: guarantees the strongest recovery by completely masking failures, so the output after failure is same as the output without failure.

Table I shows output streams of the mentioned recovery types. Each stream consists of the sequence of tuples t_i before and after the failure. As we can see, precise recovery contains a sequence that is identical to a sequence without failures. In case of gap recovery, there are some tuples missing; they create undesirable gaps in the data stream. The output of the repeatable rollback recovery contains identical tuples as in the regular output, but some of the tuples are duplicated. On the other hand, the outputs of the convergent and divergent rollback recoveries contain different tuples t'_j after failure. In case of the convergent recovery, different tuples became identical to regular tuples over time.

A. Operators Classification

We distinguish four operator types according to recovery semantics: *arbitrary*, *deterministic*, *convergent-capable* and *repeatable*. Recovery plan type is determined by the most common operator in it.

An operator is *deterministic* if it produces the same output stream every time it starts from the same initial state and receives the same sequence of tuples on each input stream. There are three possible causes of non-determinism in operators: dependence on the time, dependence on the arrival order of tuples on different input streams, and use of non-determinism in processing (e.g., randomization).

A deterministic operator is (called) *convergent-capable* if it yields a convergent recovery when it restarts from an empty initial internal state and re-processes the same input streams, starting from an arbitrary earlier point in time.

A convergent-capable operator is *repeatable* if it capable of a repeating recovery when it restarts from an empty initial internal state and re-processes the same input streams, starting from an arbitrary earlier point in time and the operator produces identical tuples.

III. HIGH AVAILABILITY PROTOCOLS

There are three basic approaches for achieving high availability in distributes systems: *process-pairs*, *logging* and *checkpointing*. In the following section, we introduce some of the current algorithms based on these approaches. Even when not all of them accomplish precise recovery, they can be extended to such level. Such extension includes for example duplicity removal and protection from data loss. Data loss protection is mostly done by logging messages in output buffers until they are processed or stored by downstream (nodes further in the data flow) nodes. In case of failure, these stored messages are resend. Duplicity removal is protocol specific, so it is mentioned separately with each method.

A. Passive standby

Passive standby [6], [5] method is based on the *process-pairs* approach. There is a secondary node assigned to each primary node that receives state updates (checkpoints) from the primary node in regular intervals. In case of failure, the secondary node takes over computation from the last checkpoint. To achieve precise recovery, it is necessary to resend the data sent by upstream nodes, to recreate failed node state on the new node and ask the downstream nodes for delivered tuples, so they are not send for the second time.

The main advantage of this method is short recovery time consisting of reprocessing tuples received since last checkpoint and discovering tuples that were already send by the crashed node. However, the computing power of the regular computation is degraded, because (at least) half of the nodes are allocated as secondary nodes and communication is increased by sending regular checkpoints. Another possible slowdown is introduced when using synchronous backup (primary node does not send data until checkpoint is confirmed on secondary). When using asynchronous backup, data loss and inconsistent state can happen until logging of output buffers is introduced.

B. Active standby

Active standby is another version of the *process-pairs* approach [6], [5], [7]. Similarly to passive standby, each processing node has a dedicated secondary node. But unlike passive standby a secondary node does actively obtain and process same tuples as a primary node. Secondary node output is then logged out instead of send further downstream. Preventing duplicate messages by identifying the messages received by downstream nodes before sending the same messages computed by the secondary node is necessary to achieve precise recovery. Also, in case of non-deterministic operators, their decisions made on primary nodes have to be logged and sent to the secondary node, so it produces exactly the same results.

Minimal recovery time is the main advantage of the active standby over passive standby. That is because there is no need to reprocess data after failure, however at the cost of significantly higher communication, because all data must be sent also to secondary nodes. Another extra communication may introduce a queue trimming protocol, and sending of the decision logs in case of non-deterministic operators.

C. Upstream backup

Large run-time overhead is the main drawback of the process-pair approach (where at least half of the nodes are designated as backup nodes, thus not actively participating in computation). Upstream backup [6], [5] is designed for better use of distributed character of a stream computation. Upstream nodes (nodes against data flow) serve as backups for their downstream nodes by logging their output tuples. In case of

 ISBN: 978-1-61208-334-6

failure, a new node with empty state takes over computation and reprocesses stored tuples to get the same internal state as the failed node had.

Output buffers that backup tuples can grow in size equivalent to the size of tuples passing through that can be very space-demanding. To solve this inefficiency, the queue trimming protocol is introduced. It is based on the finding a minimal set of stored tuples that is necessary to restore a crashed node state. The delivery confirmation protocol is used to inform upstream nodes about tuples that were delivered. Delivery of each tuple is confirmed using 0-level confirmation to the sender of the tuple. After receiving 0-level confirmation, the node will know that the given tuple and all the proceeding tuples were delivered by recipient that send the confirmation. Once a tuple is confirmed by all recipients, the node determines last input tuple that was used to compute confirmed one, but it is not used to generate newer tuples anymore. If such tuple is found, it is confirmed to its sender using 1-level confirmation. Output buffers can be trimmed at the position of a tuple that received 1-level confirmation from all its recipients. Higher confirmation levels (by iteratively repeating confirmations) can be used to achieve better protection against multiple faults, however at the cost of less trimming efficiency, thus higher data space requirements.

Low extra bandwidth that is necessary for small confirmations and data transfers only during recovery is the main advantage of this method. However, this is at the cost of re-computation of many tuples during recovery. Also fail-free computation is slowed down when computing higher level confirmations that can be non-trivial.

D. Cooperative passive standby

Cooperative passive standby [8] is based on the *checkpoint* approach and on advantages of distributed computing (i.e., expandability, improved performance, etc.). Each computation is composed of computational units that are connected by data streams. Computational units are assigned to available hardware nodes to execute them. Traditional checkpointing of a more complex, or data intensive units is time demanding. To achieve better performance, this method splits computational units on each node into smaller parts called HA units. HA units are captured independently and then backed up on different nodes. This splitting divides the backup load of one node among multiple nodes.

Backing up each HA separately introduces finer granularity of the backup task; therefore, it can better fit into the spare time during computation (for example when pending for data) and increase overall system performance. Backup of each HA unit is done in two steps - capture and paste. During the capture, the update of the HA unit state is recorded and sent to the assigned backup node. During the paste, the node takes a received state updates for HA units backed up on it and apply delta updates into its copy of HA unit image. After paste, the initial node is notified, so it can schedule the HA unit for another checkpoint.

When a failure occurs, the backup nodes take over computations of the crashed node. During the takeover, the paste operations of unprocessed update messages take place and data are redirected to the backup nodes. Also to reflect the changes that occurred between backup and crash, the tuples not included in the backups are resend from output buffers, so they can be reprocessed. Computation of HA units now continues on backup nodes.

This method has fast recovery time and the expected increase of workload on backup nodes is sufficiently small to preserve efficiency. If the crashed node becomes available again, it is added as a new empty node. HA units or their backups can be assigned to empty nodes or nodes with low workload as part of load balancing. HA units can be created and distributed between nodes automatically, so it can reflect actual situation and load balance.

IV. D-BOBOX

The Bobox [9], [3] is a parallel framework, which was designed to support development of data-intensive parallel computations. The main idea behind Bobox is to create a system that connects a large number of relatively simple computational components into a nonlinear pipeline while preserving transparency of the distribution logic to the authors of computational components. The pipeline is then executed in parallel, but the interface used by the computational components is designed in such way, that their developers do not need to be concerned with the parallel execution issues such as scheduling, synchronization and race conditions.

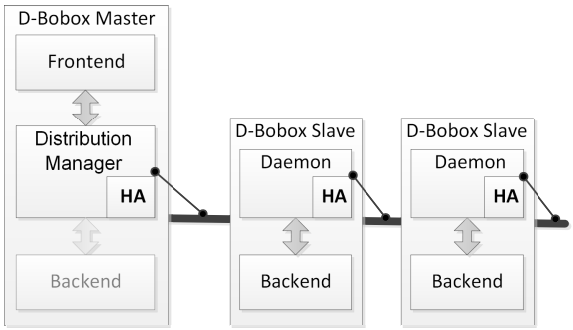

Figure 1. D-Bobox architecture. There is a single Master node that is responsible for task preparation and its distribution to slave nodes. The remote communication at the Master node is done via its Distribution Manager. Slave nodes contains Daemon parts that is responsible for the remote communication and tasks. The Distribution Manager and the Deamon are extended to provide HA. The Backend part on the Master node is optional.

D-Bobox [10] is an extension of the Bobox framework that adds support for distributed environment. This allows the framework to be used for tasks where local parallelism is not enough to achieve effectively fast computation. To preserve versatility of the framework, it is designed to run not only on specialized computational clusters, but on a common hardware too.

Base schema of the D-Bobox is described in Figure 1. The *master node* is responsible for creating an execution plan of the task and communication with the user that enters the task and monitors its computation. The master node also decides which other nodes will be participating in computation as *work (slave) nodes* and (typically) collects results. D-Bobox uses Bobox computation logic at each slave node and wraps it with remote communication and other necessary functionality needed for distributed environment. A remote communication

 ISBN: 978-1-61208-334-6

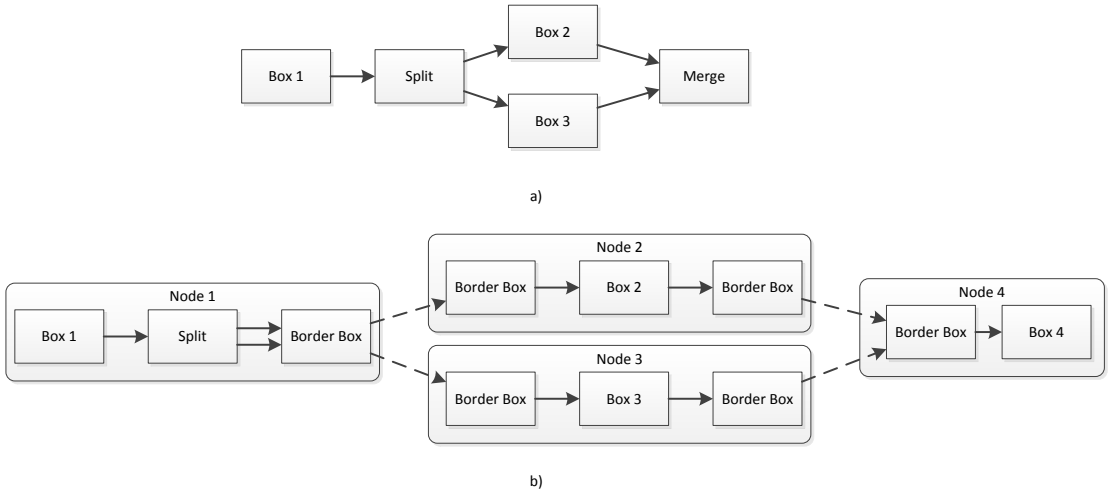

Figure 2. Sample of an execution extended into distributed environment using boundary boxes. a) Bobox plan for single node execution. b) D-Bobox plan utilizing four nodes. Original plan is split and extended by adding boundary boxes that manage remote communication (dashed).

is the most important extension during execution. It is implemented in the special boxes - border boxes that are added into the execution plan before slave nodes are initialized. Adding and configuring of border boxes is done primarily by the distribution control logic at the master node, according to the actual configuration and availability of nodes in cluster. Border boxes on a slave node are configured on request of the master node when necessary. Example extended execution plan is depicted at Figure 2.

V. HA AND D-BOBOX

A. Basic algorithms and D-Bobox

Each of the algorithms mentioned in Section III is applicable to D-Bobox system with different impacts to transparency, computation boxes requirements and changes to the framework itself. In the following discussion, we are focusing only on deterministic operators. Implementation of non-deterministic boxes breaks transparency requirement for each method, as they are required to log non-deterministic decisions and provide backup logic to achieve precise recovery.

Upstream backup represents a least blocking approach with minimum extra communication during fault-free computation. However, space requirements for storing output queues, recovery time and computation needed after the failure are typically quite high. Implementation of queue trimming protocols reduces these disadvantages at cost of higher communication and some computation slowdown during building and searching mappings between input and output tuples. When users are creating a new box, they have to implement these sometimes non-trivial mappings. That negatively impacts the transparency requirement and reliability of the framework. Reliability of the framework that depends on correct user implementation of the mappings is not suitable for our framework.

The active standby approach represents maximum transparency. It can be reflected by the computation plan used in D-Bobox by duplicating appropriate plan fragments and redirecting each of their outputs to the special communication box, located on the downstream node, so it is not affected by the failure of the primary or the secondary node. This communication box receives tuples produced by the primary and secondary node; and forwards only tuples from primary node and stores tuples from the secondary node not yet produced by the primary node.

When a node failure occurs, the secondary node then becomes primary and continues in the computation. Then new secondary node is chosen and initialized by the new primary node current state. Computation speed at the primary node can differ from the secondary node, so after a failure, the communication box must correctly manage the data stream to preserve data consistency. When the primary node was ahead of the secondary, then the communication box must drop duplicate tuples produced by the secondary to prevent duplicate data. At the opposite situation, when the primary node was slower than the secondary node, then the communication box must send stored tuples to prevent gap in the data stream.

Effective computation power of active standby is halved since half of the nodes are reserved as backup nodes. Therefore, this approach is appropriate for problems, where very fast recovery time is more critical aspect than overall computation time. Since D-Bobox is oriented to be computation efficient, this approach is not appropriate to provide base HA functionality. However, it can be easily introduced for specific tasks that require very fast recovery instead of fast processing.

Passive standby approach also suffers from the same cutting of the effective computation performance as the active standby and is slower in recovery and during regular computation. Therefore, it is less practical than active standby. Another transparent approach is Cooperative passive standby that combines checkpointing and splitting backup to multiple smaller tasks and then distribute it between distinct nodes. Distribution of the smaller tasks increases backup performance, reduces recovery time and reduces work increase on backup nodes. Thanks to its distribution character and potential efficiency, we decided to use it as the base method for recovery from node failure to achieve HA in D-Bobox system. In the following subsection, we describe its integration in more detail.

 ISBN: 978-1-61208-334-6

B. Integration of the High availability into D-Bobox

High Availability support that will handle node failures in D-Bobox is based on the cooperative passive standby method and it is located in new execution units called HA managers. HA managers are divided according to their specialization and location in the system to:

- *global HA manager* located at the primary node,
- *local HA managers* located at secondary (worker) nodes.

Each manager type handles different tasks: local HA managers perform local tasks that include local computation and backups stored on local node. The global HA manager is a global coordinator that assigns backup nodes for local HA units and handles recovery after failure detection.

1) HA manager at the primary node: is also called global HA manager. Its primary focus is to handle global tasks such as failure handling, assigning backup nodes to HA units and cooperation with the load balancing unit. An example of the distribution of backups of HA units to other nodes, as assigned by the global HA manager, is depicted in Figure 3.

Handling a node failure at the primary node is described in the Algorithm 1. Global HA unit notifies backup nodes to take over crashed node computation, reroute data and choose new backup nodes for restored HA units. After notifying upstream HA units, they resend data from output buffers to recreate the lost state not reflected in the last checkpoint. The increase of workload at a backup node after takeover is expected to be in acceptable boundaries because of the distribution of the work among multiple physically independent nodes.

A crashed node joins the set of nodes after recovery as an empty node and it can be dynamically assigned to backups or tasks during load balancing or after crash of another node. Global HA manager should support load balancing to achieve better performance of the computation and backups. When the system is highly unbalanced, then heavily loaded nodes cannot backup efficiently. They do not have spare time to backup, so they increase backup intervals that make backup more difficult and more costly, or they block computation often. On the other hand, idle nodes produce backups that can further slowdown loaded nodes. Dynamic load balancing increases the chance of evenly scheduling backups into idle CPU cycles when the computation is waiting (i.e., to receive new tuples). Moreover, processing backups more frequently reduces the amount of containing tuples in the backup, and less tuples have to be stored in the output buffers and recomputed during recovery.

2) HA manager at the secondary node: represents local manager that manage HA tasks of the current node as for example:

- monitoring neighbors availability (both upstream and downstream),
- administration of local HA units (for example splitting, merging),
- planning and performing of HA units backup (represented by the operation capture),

Algorithm 1 Processing a node crash on the primary node

```
for all HAunit in crashedNode do
  backupNode ← getBackupNode(HAunit)
  backupNode.takeOver(HAunit)
  for all edge in HAunit.io do
    if isRemote and isInput then
      set_edge_target_to_backup
      resend_cached_tuples
    else if isRemote and isOutput then
      set_edge_source_to_backup
    else if isLocal and isInput then
      add_new_remote_edge
    end if
  end for
end for
for all backup in CrashedNodeBackups do
  find_and_set_new_backup_node
end for
```

- planning and performing of merge of the received updates from remote HA units into their local backups (represented by the operation paste).

Availability of neighbors is monitored by each node. Requests sent in regular intervals to test the availability are used when there is no communication among nodes at the time. If the node stops to respond, then after a defined amount of time (according to a preset time limit) is declared as crashed. The global HA manager is notified that node failure occurs and left to take care of the situation.

Capture and *paste* operations are two main operations providing backup functionality. During capture operation, the difference in HA unit state is captured and sent to the backup node. Paste operation on the backup node processes the received update messages by applying them to the local copy of HA unit state. Capture and paste operations of each HA unit are performed independently of each other, according to the used scheduling algorithm. Scheduling is performed locally on each node by the local scheduler that can implement different strategies to balance backup performance and computation performance.

Local HA managers support splitting and joining local HA units to balance granularity. Splitting is possible only if a HA unit consists of more than one computation box. Typically splitting a more complex HA unit results in dividing the backup overhead into a few smaller units that are easier to backup. Smaller HA units also pose smaller increase of the workload on the backup node, when it takes over the computation after recovery. On the other hand, a join operation is used to group multiple simple operations where an increase in the backup cost is smaller than the reduction of backup communication. For example, few HA units, each consisting of a simple box connected together into a pipeline, are good candidates to merge. In this case, we can suppose that the increase of backup complexity of the joined HA unit will be smaller (simple boxes, locality of the data) than the backup overhead of multiple HA units alone.

These changes of HA units size granularity have to be coordinated with the global HA manager. It must assign new

 ISBN: 978-1-61208-334-6

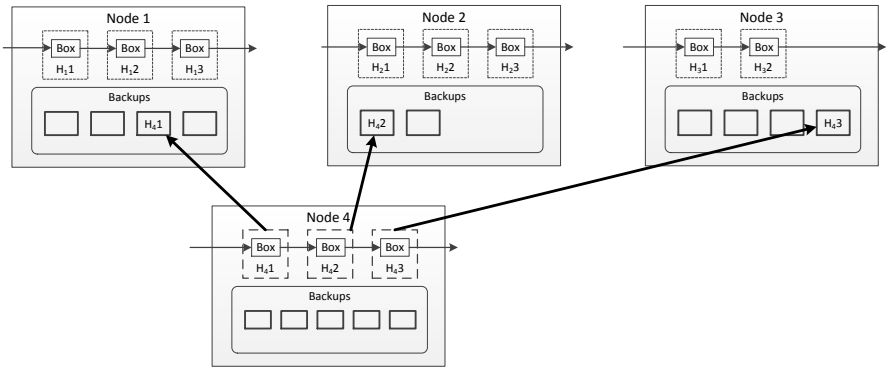

Figure 3. Example of the distribution of HA units demonstrated on the Node 4. Execution plan of the Node 4 is split into three HA units $H_4 1$,..,$H_4 3$ that are backed up to the nodes 1-3 (backups are represented as boxes $H_4 1$,..,$H_4 3$ in backup part of nodes 1 to 3).

backup nodes to each new HA unit that was created by the split operation or revoke backup role from nodes in case of joining of HA units.

A secondary node also contains a specific local communication manager that is part of special boundary boxes. Boundary boxes are special system boxes that represent endpoints for the remote communication to hide it from users developing regular computation boxes. Communication HA manager extends them by adding message logging and duplicity elimination. Outgoing messages are logged until the confirmation of their backup from HA units arrive. This is necessary to restore the computation state by computing these messages again after recovery from the last backup to reflect lost changes. Input border boxes have to be able to eliminate duplicity tuples that may be produced during recovery (crashed node may or may not produce some tuples that are not reflected in backup).

Extending D-Bobox with proposed managers provides high availability support to the framework. By using Cooperative passive standby as a core method that splits the node tasks into separate HA units distributed to different nodes, we get efficient recovery after a possible node failure. Another fine tuning of the HA units granularity according to actual computation states further improves the overall system performance. Smaller HA unit backups also pose acceptable increase of workload of the backup nodes after the recovery. The proposed approach also preserves transparency; the users creating applications should not be concerned with recovery methods.

VI. Conclusion and Future Work

In this paper, we introduced our distributed framework D-Bobox that is targeted on processing BigData (i.e., semantic databases). We presented a node failure problem in distributed data stream processing systems. Then we defined recovery types and summarized the main contemporary approaches to achieve high availability in such systems. We analyzed these approaches for their applicability in D-Bobox; we proposed an implementation of HA support in the framework. Using such HA support, the framework became capable of creating failure-resistant applications for data intensive computations in the distributed environment on a commodity hardware. The framework also preserves high level of transparency; the users do not have to solve technical details concerning parallelism, distributed processing or high availability logic. In our future work, recovery support can be further extended by adding support of nondeterministic operators or adding new scheduling or by load balancing strategies.

Acknowledgment

The authors would like to thank the GAUK project no. 472313 and SVV-2014-260100 and GACR project no. P103/13/08195S, which supported this paper.

References

[1] M. Balazinska *et al.*, "The design of the borealis stream processing engine," in *Second Biennial Conference on Innovative Data Systems Research (CIDR 2005)*. CIDR, 2005, pp. 277–289.

[2] D. Abadi *et al.*, "Aurora: a data stream management system," in *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. ACM, 2003, pp. 666–666.

[3] D. Bednarek, J. Dokulil, J. Yaghob, and F. Zavoral, "Bobox: Parallelization Framework for Data Processing," in *Advances in Information Technology and Applied Computing*, 2012, pp. 189–194.

[4] Z. Falt, J. Dokulil, M. Cermak, and F. Zavoral, "Parallel sparql query processing using bobox," *International Journal On Advances in Intelligent Systems*, vol. 5, no. 3, pp. 302–314, 2012.

[5] J.-H. Hwang *et al.*, "High-availability algorithms for distributed stream processing," in *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*, 2005, pp. 779–790.

[6] J.-H. Hwang, M. Balazinska *et al.*, "A comparison of stream-oriented high-availability algorithms," Brown CS, Tech. Rep., 2003.

[7] M. A. Shah, J. M. Hellerstein, and E. Brewer, "Highly available, fault-tolerant, parallel dataflows," in *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, ser. SIGMOD '04. New York, NY, USA: ACM, 2004, pp. 827–838. [Online]. Available: http://doi.acm.org/10.1145/1007568.1007662

[8] J.-H. Hwang, Y. Xing, U. Cetintemel, and S. Zdonik, "A cooperative, self-configuring high-availability solution for stream processing," in *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*. IEEE, 2007, pp. 176–185.

[9] D. Bednrek, J. Dokulil, J. Yaghob, and F. Zavoral, "Data-flow awareness in parallel data processing," in *Intelligent Distributed Computing VI*, ser. Studies in Computational Intelligence, G. Fortino, C. Badica, M. Malgeri, and R. Unland, Eds. Springer Berlin Heidelberg, 2013, vol. 446, pp. 149–154.

[10] M. Cermak, Z. Falt, and F. Zavoral, "D-bobox: O distribuovatelnosti boboxu," in *Informacne Technologie - Aplikacie a Teoria*. PONT s. r. o., 2012, pp. 41–46.

 ISBN: 978-1-61208-334-6