Fault-tolerance and high availability in data stream management systems

Magdalena Balazinska
University of Washington
http://www.cs.washington.edu/homes/magda/

Jeong-Hyon Hwang
Brown University
http://www.cs.brown.edu/people/jhhwang/

Mehul A. Shah
HP Labs
http://www.hpl.hp.com/personal/Mehul_Shah/

SYNONYMS

None

DEFINITION

Just like any other software system, a data stream management system (DSMS) can experience failures of its different components. Failures are especially common in *distributed* DSMSs, where query operators are spread across multiple processing nodes, i.e., independent processes typically running on different physical machines in a local-area network (LAN) or in a wide-area network (WAN). Failures of processing nodes or failures in the underlying communication network can cause continuous queries (CQ) in a DSMS to stall or produce erroneous results. These failures can adversely affect critical client applications relying on these queries.

Traditionally, availability has been defined as the fraction of time that a system remains operational and properly services requests. In DSMSs, however, availability often also incorporates end-to-end latencies as applications need to quickly react to real-time events and thus can tolerate only small delays. A DSMS can handle failures using a variety of techniques that offer different levels of availability depending on application needs.

All fault-tolerance methods rely on some form of replication, where the volatile state is stored in independent locations to protect against failures. This article describes several such methods for DSMSs that offer different trade-offs between availability and runtime overhead while maintaining consistency. For cases of network partitions, it outlines techniques that avoid stalling the query at the cost of temporary inconsistency, thereby providing the highest availability. This article focuses on failures within a DSMS and does not discuss failures of the data sources or client applications.

HISTORICAL BACKGROUND

Recently, DSMSs have been developed to support critical applications that must quickly and continuously process data as soon as it becomes available. Example applications include financial stream analysis and network intrusion detection (see KEY APPLICATIONS for more). Fault-tolerance and high availability are important for these applications because faults can lead to quantifiable losses. To support such applications, a DSMS must be equipped with techniques to handle both node and network failures.

All basic techniques for coping with failures involve some kind of replication. Typically, a system replicates the state of its computation onto independently failing nodes. It must then coordinate the replicas in order to recover properly from failures. Fault-tolerance techniques are usually designed to tolerate up to a pre-defined number, k, of simultaneous failures. Using such methods, the system is then said to be k-fault tolerant.

There are two general approaches for replication and coordination. Both approaches assume that the computation can be modeled as a deterministic state-machine [4, 11]. This assumption implies that two non-faulty computations that receive the same input in the same order will produce the same output in the same order. Hereafter, two computations are called *consistent* if they generate the same output in the same order.

The first approach, known as the *state-machine* approach, replicates the computation on $k + 1 \geq 2$ independent nodes and coordinates the replicas by sending the same input in the same order to all [11]. The details of how

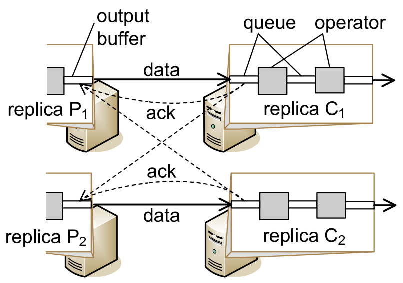

Figure 1: Active Replicas. The operators on replicas P_1 and P_2 are the producers. The operators on C_1 and C_2 are the consumers.

to deliver the same input define the various techniques. Later sections in this article describe variants that are specific to DSMSs. The state-machine approach requires $k + 1$ times the resources of a single replica, but allows for quick fail-over, so a failure causes little disruption to the output stream. This property is important for critical monitoring tasks such as intrusion detection that require low-latency results at all times.

The second general approach is known as *rollback recovery* [4]. In this approach, a system periodically packages the state of its computation into a *checkpoint*, and copies the checkpoint to an independent node or a non-volatile location such as disk. Between checkpoints, the system logs the input to the computation. Since disks have high latencies, existing fault-tolerance methods for DSMSs copy the checkpointed state to other nodes and maintain logs in memory. Upon failure, the system reconstructs the state from the most recent checkpoint, and replays the log to recover the exact pre-failure state of the computation. This approach has much lower runtime overhead at the expense of higher recovery time. It is useful in situations where resources are limited, the state of the computation is small, fault-tolerance is important, but rare moderate latencies are acceptable. An example application is fabrication-line monitoring using a server cluster with limited resources.

In some cases, users are willing to tolerate temporary inconsistencies to maintain availability at all times. One example is in the wide-area where network partitions are likely (e.g., large-scale network and system monitoring). To maintain availability in face of network partitions, the system must move forward with the computation ignoring the disconnected members. In this case, however, replicas that process different inputs will have inconsistent states. There are two general approaches for recovering from such inconsistencies after the network partition heals. One approach is to propagate all updates to all members and apply various rules for reconciling conflicting updates [6, 9]. The other approach is to undo all changes performed during the partition and redo the correct ones [6, 15].

This article presents how these general approaches can be adapted to distributed DSMSs. The main challenge is to ensure that applications receive low-latency results during both normal processing and failures. To do so, the methods presented leverage the structure of continuous queries (CQs) in DSMSs.

This article makes the following assumptions. A CQ is a connected directed-acyclic graph of query operators. The operators can be distributed among many processing nodes with possibly multiple operators per node. A processing node is the unit of failure. For simplicity of exposition, this article focuses on the case of $k = 1$ (i.e., two query replicas) although all shown techniques can handle any k. This article describes methods to tolerate node crash failures and temporary network partitions. Roughly speaking, a node has crashed if it visibly halts or simply becomes unresponsive [12]. A network partition splits nodes into two or more groups where nodes from one group cannot communicate with nodes in another.

SCIENTIFIC FUNDAMENTALS

Techniques for Handling Crash Failures

This section describes new fault-tolerance techniques devised by applying the general fault-tolerance methods to continuous queries in DSMSs.

Active Replicas

Active replicas are an application of the state-machine approach in which query operators are replicated and run

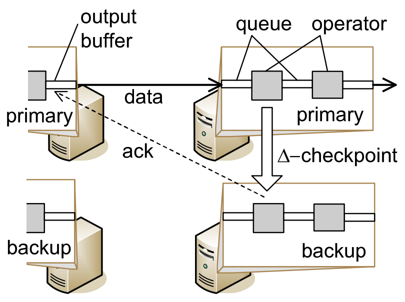

Figure 2: Passive Standby

on independently failing nodes. A simple variant of the active replicas approach uses the traditional process-pair technique to coordinate the replicas. The process-pair technique runs two copies of the query and specifies one to be the primary and the other to be the secondary. In this approach, the primary forwards all input, in the same order, to the secondary and works in lock-step with the secondary [5].

A DSMS can rely on a looser synchronization between the replicas by taking advantage of the structure of CQ dataflows. In a CQ dataflow, the operators obey a producer-consumer relationship. To provide high availability, the system replicates both the producer and consumer as illustrated in Figure 1. In this model, there is no notion of a primary or secondary. Instead, each producer logs its output and forwards the output to its current consumer(s). Each consumer sends periodic acknowledgments to all producers to indicate that it has received the input stream up-to a certain point. An acknowledgment indicates that the input need not be resent in case of failure, so producers can truncate their output logs. Use of reliable, in-order network delivery (e.g., TCP) or checkpoints allows optimizations where consumers send application-level acknowledgments to only a subset of producers [7, 13].

The symmetric design of active replicas has some benefits. The normal-case behavior has few cases, so it is simple to implement and verify. Additionally, with sufficient buffering, each pipeline can operate at its own pace, in looser synchronization with the other.

The Flux [13] approach was the first to investigate this looser synchronization between replicated queries. Flux is an opaque operator that can be interposed between any two operators in a CQ. Flux implements a simple variant of this protocol and assists in recovery. The Borealis "Delay, Process, and Correct" (DPC) protocol [1, 2] also uses the above coordination protocol, but differs from Flux in its recovery, as discussed later. The Flux and DPC approaches both ensure strict consistency in the face of crash failures: no duplicate output is produced and no output is lost.

Passive Replicas

There have been two applications of the rollback recovery approach to CQs [7, 8]. The first, called *passive standby*, handles all types of operators. The second, called *upstream backup*, is optimized for more specific bounded-history operators that frequently arise in CQs.

In the passive standby approach, a primary node periodically checkpoints its state and sends that checkpoint to a backup. The state includes any data maintained by the operators and tuples stored in queues between operators. In practice, sending the entire state at every checkpoint is not necessary. Instead, each primary periodically performs only a *delta-checkpoint* as illustrated in Figure 2. During a delta-checkpoint, the primary updates the backup by copying only the difference between its current state and the state at the time of the previous checkpoint.

Because of these periodic checkpoints, a backup always has its primary's state as of the last checkpoint. If the primary fails, the backup recovers by restarting from that state and reprocessing all the input tuples that the primary processed since the last checkpoint. To enable backups to reprocess such input tuples, all primaries log their output tuples. If a downstream primary fails, each upstream primary re-sends its output tuples to the downstream backup. In a CQ, because the output of an operator can be connected to more than one downstream consumer operator, primaries discard logged output tuples only after *all* downstream backups have acknowledged a checkpoint.

For many important CQ operators, the internal state often depends only on a small amount of recent input.

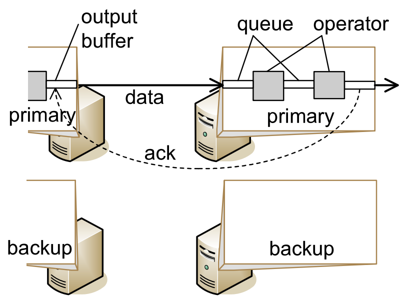

Figure 3: Upstream Backup

Examples of such operators include joins and aggregates with windows that span a short time-period or a small number of tuples. For such operators, DSMSs can use the upstream backup method to avoid any checkpointing overhead. In this approach, primaries log their output tuples, but backups remain idle as illustrated in Figure 3. The primaries trim their logs based on notifications from operators 1-level (or more) downstream, indicating that the states of consuming operators no longer depend on the logged input. To generate these notifications, downstream operators determine, from their output, what logged input tuples can be safely discarded. If a primary fails, an empty backup rebuilds the latest state of the primary using the logs kept at upstream primaries.

Failure Recovery

When a failure occurs, a DSMS must first detect and then recover from that failure. DSMSs detect failures using timeouts and, in general, rely on standard group membership mechanisms to keep consistent track of nodes entering and leaving the system [10, 14]. Recovery ensues after failure detection.

There are two parts to failure recovery. The first part involves masking the failure by using the remaining replica to continue processing. For active replicas, this part is called *fail-over*. In both Flux and DPC, fail-over is straightforward. Consumers and producers adjust their connections to receive input data from or send output data to the remaining live copy of the failed node. To avoid stalls in the output stream, it is safe for active replicas to proceed with fail-over without waiting for group membership consensus [1, 14]. For passive standby and upstream backup, this first part also involves bringing the state of the backup to the pre-failure state of the failed primary, as described earlier, before the backup starts sending data to downstream consumers.

The second part of recovery, called *repair*, allows the query to repair its failed pieces and regain its original level of fault-tolerance. In upstream backup, the system regains its normal fault-tolerance level when the new replica fills its output log with enough data to rebuild the states of downstream nodes.

For both active replicas and passive standby, repair can cause significant disruptions in the result stream depending on the granularity of coordination in the query. For example, if a system uses active replica coordination only at the input(s) and output(s) of a distributed query, the system must destroy the *entire* query affected by the failure, stall the *entire* remaining query, checkpoint its state, copy that state onto independent nodes, and reintegrate the new copy with the remaining query. The system must repair a query at a time because it has no control over inflight data in the network between nodes in a query. If the query state is large, e.g. tens of gigabytes, repair can take minutes, causing significant latencies in the result stream. Similarly, coarse coordination in Passive Standby would cause the first checkpoint after recovery to stall processing for a long time.

To remedy this problem, most high-availability CQ schemes (e.g. Flux [13, 14], Borealis DPC [1, 2], Active Standby [7], Passive Standby [7, 8]) coordinate and repair in smaller chunks: between nodes (containing groups of operators), between operators, or even finer. Then, after failure, they can repair the lost pieces one at time, allowing the remaining pieces to continue processing and reduce the impact of stalls. In the presence of $k+1 > 2$ replicas, DSMSs can use the extra replicas to further smooth the impact of stalls during repair. Also with finer coordination, DSMSs need to repair only the lost pieces, thereby reducing mean-time-to-recovery and improving system reliability, i.e. mean-time-to-failure [13].

Trade-offs Among Crash failure Techniques

The above techniques provide different trade-offs between runtime overhead and recovery performance. Active replicas provide quick fail-over because replicas are always "up-to-date". With this approach, however, the runtime overhead is directly proportional to the degree of replication. Passive standby provides a flexible trade-off between runtime overhead and recovery speed through the configurable checkpoint interval. As the checkpoint interval decreases, the runtime computation and network overheads increase because the primaries copy more intermediate changes to the backups. However, recovery speed improves because the backups are in general more up-to-date when they take over. Finally, upstream backup incurs the lowest overhead because backups remain idle in the absence of failures. For upstream backup, recovery time is proportional to the size of the upstream buffers. The size of these buffers, in turn, depends on how much history is necessary to rebuild the state of downstream nodes. Thus, upstream backup is practical in small history settings.

Techniques for Handling Network Partitions

The previous techniques can mask crash failures and a limited set of network failures (by converting disconnected nodes into crashed nodes), but cannot handle network partitions in which data sources, processing nodes, and clients are split into groups that cannot communicate with each other.

In the presence of network partitions, a DSMS, like all distributed systems, has two choices. It can either suspend processing to ensure consistency, or continue processing the remaining streams with best-effort results to provide availability [3]. Existing work on fault-tolerance in distributed DSMSs has explored both options. The Flux protocol [14], originally set in the local-area where network partitions are rare, favors consistency. The Borealis's DPC protocol, designed for wide-area monitoring applications where partitions are more frequent, favors availability. During partitions, DPC generates best-effort result tuples which are labeled as *tentative*. Further, DPC allows applications to specify a maximum tolerable latency for flexibly controlling the tradeoff between consistency and availability [1, 2].

Once a network partition heals, a stalled CQ node can simply resume. A node that continued processing with missing input, however, might be in a diverged state, i.e., a state different from that of a failure-free execution. To reconcile a node's diverged state, a DSMS can take two approaches. The system can revert the node to a consistent, checkpointed state and replay the subsequent, complete input, or the system can undo and redo the processing of all tuples since the network partition. To avoid stalling the output during reconciliation, a DSMS must take care not to reconcile all replicas of a node simultaneously. Moreover, nodes must correct their previous output tentative tuples to enable downstream nodes to correct their states, in turn, and to allow applications to ultimately receive the correct and complete output. The Borealis DPC protocol supports these techniques [1, 2].

Optimizations

Flux: Integrating Fault Tolerance and Load Balancing

A large-scale cluster is a dynamic environment in which DSMSs face not only failures but also load imbalances which reduce availability. In this setting, DSMSs typically split the state of operators into *partitions* and spread them across a cluster for scalability. A single overloaded machine, in this setup, can severely slow down an entire CQ. The Flux operator can be interposed between producer-consumer partitions to coordinate their communication. To absorb short delays and imbalances, Flux allows out-of-order processing of partition input. For long-term imbalances, it supports fine-grained, online partition state movement. The Flux operators interact with a global controller that coordinates repair and rebalancing, and uses the same state movement mechanism for both [14]. Integrating load balancing with fault tolerance allows a system to better utilize available resources as nodes enter and leave the system. These features allow smooth hardware refresh and system growth, and are essential for administering DSMSs in highly dynamic and heterogeneous environments.

Leveraging Replication for Availability and Performance in Wide-Area Networks

In previous active replicas approaches, a consumer replica receives the output stream of only one of many producer replicas. In wide area networks, the connection to any single producer is likely to slow down or fail, thereby disrupting the subsequent processing until fail-over completes. To avoid such disruptions, each consumer replica in Hwang et al.'s method [8] merges streams from multiple producer replicas into a single duplicate-free stream. This scheme allows each consumer, at any instant, to use the fastest of its replicated input streams. To further reduce latency, they redesign operators to avoid blocking by processing input out-of-order when possible, while ensuring that applications receive the same results as in the non-replicated, failure-free case. Moreover, their scheme continually adjusts the replication level and placement of operator replicas to optimize global query performance in a dynamically changing environment.

Passive Standby: Distributed Checkpointing and Parallel Recovery

The basic passive standby approach has two drawbacks: it introduces extra latencies due to checkpoints and has a slower recovery speed than active replicas. Hwang et. al.'s distributed checkpointing technique overcomes both problems [8]. This approach groups nodes into logical clusters and backs up each node using the others in the same cluster. Because different operators on a single node are checkpointed onto separate nodes, they can be recovered in parallel. This approach dynamically assigns the backup node for each operator and schedules checkpoints in a manner that maximizes the recovery speed. To reduce disruptions, this approach checkpoints a few operators at a time. Such checkpoints also begin only at idle times.

KEY APPLICATIONS

There are a number of critical, online monitoring tasks that require 24x7 operation. For example, IT administrators often want to monitor their networks for intrusions. Brokerage firms want to analyze quotes from various exchanges in search for arbitrage opportunities. Phone companies want to process call-records for correct billing. Web site owners want to analyze and monitor click-streams to improve targeted advertising and to identify malicious users. These applications, and more, can benefit from fault-tolerant and highly available CQ systems.

FUTURE DIRECTIONS

Key open problems in the area of fault-tolerance and high availability in DSMSs include handling Byzantine failures, integrating different fault-tolerance mechanisms, and leveraging persistent storage. Techniques for handling failures of data sources or dirty data produced by data sources (e.g., sensors) are also areas for future work.

EXPERIMENTAL RESULTS

See [1, 2, 7, 8, 13, 14] for detailed evaluations of the different fault-tolerance algorithms.

DATA SETS

URL TO CODE

Borealis is available at: `http://www.cs.brown.edu/research/borealis/public/`

CROSS REFERENCE

Data stream management system (DSMS), distributed DSMS, continuous query (CQ).

RECOMMENDED READING

[1] Magdalena Balazinska. *Fault-Tolerance and Load Management in a Distributed Stream Processing System.* PhD thesis, Massachusetts Institute of Technology, February 2006.

[2] Magdalena Balazinska, Hari Balakrishnan, Samuel Madden, and Michael Stonebraker. Fault-tolerance in the Borealis distributed stream processing system. In *Proc. of the 2005 ACM SIGMOD International Conference on Management of Data*, June 2005.

[3] Eric A. Brewer. Lessons from giant-scale services. *IEEE Internet Computing*, 5(4):46–55, 2001.

[4] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, 2002.

[5] Jim Gray. Why do computers stop and what can be done about it? *Technical Report 85.7, Tandom Computers*, 1985.

[6] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. In *Proc. of the 1996 ACM SIGMOD International Conference on Management of Data*, June 1996.

[7] Jeong-Hyon Hwang, Magdalena Balazinska, Alexander Rasin, Uğur Çetintemel, Michael Stonebraker, and Stan Zdonik. High-availability algorithms for distributed stream processing. In *Proc. of the 21st International Conference on Data Engineering (ICDE)*, April 2005.

[8] Jeong-Hyon Hwang, Ying Xing, Uğur Çetintemel, and Stan Zdonik. A cooperative, self-configuring high-availability solution for stream processing. In *Proc. of the 23rd International Conference on Data Engineering (ICDE)*, April 2007.

[9] Leonard Kawell, Steven Beckhardt, Timothy Halvorsen, Raymond Ozzie, and Irene Greif. Replicated document management in a group communication system. In *Proc. of the 1988 ACM conference on computer-supported cooperative work (CSCW)*, September 1988.

[10] Andre Schiper and Sam Toueg. From set membership to group membership: A separation of concerns. *IEEE Trans. on Dependable and Secure Computing*, 3(1):2–12, 2006.

[11] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.

[12] Fred B. Schneider. What good are models and what models are good? In *Distributed Systems*, pages 17–26. ACM Press/Addison-Wesley Publishing Co, 2nd edition, 1993.

[13] Mehul Shah, Joseph Hellerstein, and Eric Brewer. Highly-available, fault-tolerant, parallel dataflows. In *Proc. of the 2004 ACM SIGMOD International Conference on Management of Data*, June 2004.

[14] Mehul A. Shah. *Flux: A Mechanism for Building Robust, Scalable Dataflows.* PhD thesis, University of California, Berkeley, December 2004.

[15] Douglas B. Terry, Marvin Theimer, Karin Petersen, Alan J. Demers, Mike Spreitzer, and Carl Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proc. of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, December 1995.