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 servicing 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 query state is stored in multiple, independent locations to protect against failures. This article describes several such methods 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 applications that must quickly and continuously process data as soon as it becomes available. An important subset of these applications include critical, online monitoring tasks that require 24x7 operation. Such tasks can be found in a variety of settings. For example, IT administrators often want to monitor their networks for intrusions. Web site owners want to analyze and monitor click-streams to improve targeted advertising and to identify malicious users. Brokerage firms want to analyze quotes from various exchanges in search for arbitrage opportunities. Phone companies want to process call-records for correct billing. In all of these cases, fault-tolerance and high availability are important because faults can lead to quantifiable losses. In order 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, 16]. 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 running computation onto $k + 1 \geq 2$ independent nodes and coordinates the replicas by sending the same input in the same order to all [16]. The details of how 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 the 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 suspends its computation, packages it 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, but incurs higher recovery times. 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, 10]. The other approach is to undo all changes performed during the partition and redo the correct ones [6, 20].

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. 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.

SCIENTIFIC FUNDAMENTALS

Types and Impact of Failures

Fault-tolerance techniques assume a failure model for processing nodes and communication links. How well these models correspond to real system failures determines the utility of the fault-tolerance method in a given setting. Failures in distributed DSMSs can arise unexpectedly from a number of sources including software bugs, hardware errors, network faults, and human errors. For most distributed systems including DSMSs, the following failure models are common and useful.

Processing node failures:

- **Fail-stop failures**. Processing nodes fail by stopping execution and losing their internal volatile state (e.g., everything that was in memory). Other nodes in the system can easily detect the fail-stop failure of a node.
- **Crash failures**. In this model, a failed processing node forever stops sending messages and stops responding to any requests (e.g. a runaway computation). Crash failures may not be detectable by other nodes [17]. This model is a superset of, and thus more widely applicable than, the fail-stop model.
- **Byzantine failures**. Processing nodes fail by exhibiting arbitrary behavior [12]. For example, erroneous output by a node (e.g. due to a buffer overflow) is considered a Byzantine failure. Although existing DSMSs have not yet addressed Byzantine failures, well-known techniques for handling these failures exist [13]. Because this model is the most general of the three, these techniques impose higher overheads than those for other failure models.

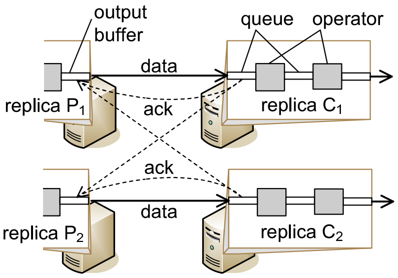

Figure 1: Active Replicas. The operators on replicas P_1 and P_2 are the producers. The operators on C_1 and C_2 are the consumers.

Communication channel failures:

In general, communication channels are unreliable and can cause message losses, re-ordering, and delays. Existing fault-tolerance techniques for DSMSs, however, typically rely on reliable, in-order communication protocols like TCP for transferring data between processing nodes. With TCP, data is always delivered reliably and in order, but the delivery can incur arbitrary delays (in the case of network failures or simply congestion) and connections can also fail permanently. **Network failures**, which cause messages to be arbitrarily delayed or TCP connections to go down, can occur in a LAN but are more common in WAN environments. Network failures can sometimes cause **network partitions**, where the entire system is split into two or more groups of nodes that can communicate with each other but cannot reach nodes in other partitions.

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 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, 18].

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

The Flux [18] 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, 9]. The first, called *passive standby*,

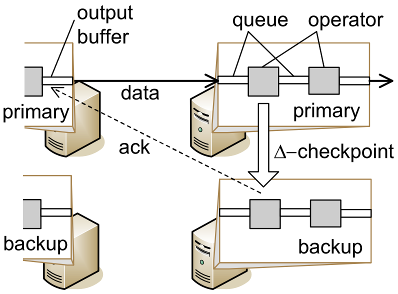

Figure 2: Passive Standby

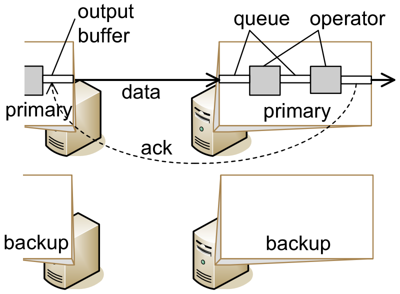

Figure 3: Upstream Backup

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.

Since latency is a critical concern in stream processing, the main challenge in passive standby is to make checkpointing as non-disruptive as possible. Clearly, independent queries can be checkpointed separately to avoid unnecessarily stalling their output. A sweep line technique can further avoid the suspension of processing. This technique moves a sweep line, in a topological order, from the input operators to the output operators of the primary. It checkpoints operators on the line while running other operators. Executing operators upstream (downstream) from the line corresponds to executing them after (before) the checkpoint.

For many important CQ operators, the internal state often depends only on a small amount of recent input. 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. The next three paragraphs describe failure recovery. How a DSMS detects failures is discussed in a later section.

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. 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 the processing for a long time. To remedy this problem, most high-availability CQ schemes (e.g. Flux [18, 19], Borealis DPC [1, 2], Active Standby [7], Passive Standby [7, 9]) 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. Finer coordination also improves system reliability [18].

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 level 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 Failures and Partitions

The above failure handling techniques are designed to enable a distributed DSMS to survive crash failures of processing nodes. These techniques can also handle a limited number of network failures by treating disconnected nodes as crashed nodes (the failure detection section below further discusses this issue). There are, however, certain types of failures that these techniques cannot mask. One interesting example is a network partition, where the data sources, processing nodes, and clients are split into two or more groups that can no longer communicate with each other.

Availability and Consistency Trade-offs during Network Partitions

In the presence of network partitions, the designer of any replication system faces a choice between providing availability or data consistency across the replicas. This trade-off is well-known in the distributed systems literature [3]. In the case of a DSMS, a network partition can cause a processing node to lose communication with all replicas that produce one of its input streams. In this case, the node can either (1) suspend all processing, ensuring that it maintains consistency or (2) continue processing the remaining input streams, providing availability but producing only best-effort results different from those of an execution without failure.

The correct choice between these two approaches depends on the application. Existing work on fault-tolerance in distributed DSMSs has explored both options. The Flux protocol [18] favors consistency when network partitions occur, while Borealis's DPC protocol [1, 2] favors availability. A system can also give applications the ability to set the trade-off between availability and consistency by specifying the maximum amount of time they are willing to wait for results. The system can then strive to minimize inconsistency, while maintaining the required low-latency processing bound [1, 2]. In the DPC protocol, nodes label tuples as *tentative* when these tuples are best-effort results produced for the sake of availability during failures.

Reintegration after a Network Partition

When a network partition heals, a processing node that was previously missing one of its input streams can now communicate again with at least one replica producing that stream. If the system favored consistency during the partition, the node was previously blocked and can now simply resume processing its input data. If the system favored availability, however, the node proceeded with a missing input stream and its state diverged during the failure. To regain a consistent state once the failure heals, replicas must thus reconcile their states.

Because no replica may have the correct state after a failure heals and because the state of a node depends on the exact sequence of tuples it processes, one approach to reconciling the state of a node is to revert it to a pre-failure state and have the node reprocess all input tuples since then. Different techniques are possible: the node can either restart from a checkpoint taken just before the failure (taken right before the node decided to proceed in spite of missing input data) or it can undo and redo the processing of all tuples [1, 2]. Both techniques incur the overhead of buffering tuples during failures in order to re-process them during state reconciliation. Furthermore, because a node that is reconciling its state is not available, extra care is required to ensure that not all replicas reconcile their states at the same time [1, 2].

As part of state reconciliation, processing nodes must also correct the tentative output tuples they produced during the failure to enable downstream nodes to correct their states in turn. These corrections also enable applications to eventually receive the correct and complete output streams.

Failure Detection

In the fail-stop model, by definition, nodes visibly fail and do not return. In this model, failure detection is trivial: each node can directly check the failure of another. Although fail-stop failures are sometimes assumed, making systems fail-stop requires extra machinery [15]. The crash failure model is thus more generally applicable. With crash failures and network failures, however, failure detection is more challenging.

In asynchronous systems (i.e., systems that make no assumptions about processing node execution speeds and message delivery delays), it is not possible to tell the difference between a slow node, a crashed node, and a disconnected node. The system can thus never be certain that a node or network failure has actually occurred. Distributed DSMSs are designed to handle asynchronous behavior including long message delays due to network congestion or slow down of processing nodes due to transient load. Such long delays are not frequent in practice. To make progress and ensure low latency results, DSMSs thus typically treat long delays as failures.

To detect failures, nodes exchange special *keep-alive* messages to check each other's health and the health of the underlying communication channels. If a node Y does not respond to a keep-alive from another probing node X within a certain time period, known as a *time out*, then X *suspects* that Y has failed. Y, however, may still be alive. It may simply be slow to respond, messages from it may be delayed, or it may be disconnected from X (though not necessarily from other nodes in the system). When multiple nodes must coordinate to act upon a failure, a node cannot always unilaterally act on a suspicion since only some, but not all, nodes may believe the suspected node has failed. Actions taken during this disagreement can lead to inconsistencies. For example, consider a passive backup that unilaterally takes over on a suspicion that the primary failed. If the suspicion is due to a temporary network failure, the primary may still be alive but simply disconnected from the backup. In that case, after recovery, both the primary and backup could send output to the same downstream consumer, causing it to erroneously receive duplicate data.

To resolve this problem when a node is suspected to have failed, the typical approach is to rely on a *group membership* protocol that removes the node from a globally agreed-upon set of nodes participating in a query. Underlying group membership algorithms is a distributed agreement protocol that consistently updates this globally agreed-upon set [11, 14]. The distributed agreement protocol can be executed directly by the nodes or by a separate query controller. This controller is itself a replicated state-machine that maintains all the metadata about the deployment of a distributed query. Flux [18] uses the controller approach for coordinating repair.

Reaching agreement on membership sometimes can take time, for example when the network is unstable. Fortunately, for high-availability applications that rely on active replicas, a DSMS need not wait for agreement to proceed with fail-over. Since both replicas are active, it is safe for a consumer to unilaterally switch to a different replica of the producer either on a suspicion or purely for performance [1, 2, 8]. The Flux and Borealis DPC protocols use this approach for quick fail-over.

Optimizations

Flux: Integrating Fault-Tolerance and Load-Balancing

In a large-scale cluster setting, a DSMS faces a few challenges: achieving scalability, handling node failures, and handling load imbalances. DSMSs typically split the state of operators into *partitions* and spread them across a cluster to achieve scalability. The Flux operator, originally designed for this setting, can be interposed between producer-consumer partitions to provide low-overhead, fine-grained coordination for these partitions.

This fine-grained coordination allows both fine-grained repair and state movement, useful for handling failures and load-balancing [19]. Flux operators work with a controller to orchestrate repair. The controller sweeps through a CQ in topological order, from input to output, repairing one partition at a time. This approach dilates recovery but avoids hiccups in the output stream. Flux also embeds load-balancing logic into the controller, so partitions can be smoothly redistributed by artificially failing overloaded partitions and repairing them onto other nodes. Combining the two allows a system to fully take advantage of available resources as nodes enter and leave the system. Thus, these features not only handle failure and repair gracefully, but also allow smooth hardware refresh and system growth. This combination of features is essential for administering and evolving DSMSs in highly dynamic and heterogeneous environments.

Aggressive Replication for Fast and Highly-Available Processing in Wide-Area Networks

In the "active replicas" approaches discussed so far, a consumer replica can receive inputs from only one of the producer replicas. Thus, the processing gets delayed if such a producer replica fails (or gets overloaded/disconnected). Furthermore, this problem may persist until each consumer replica notices it and acquires a new input connection from another functioning producer replica. Hwang et. al.'s replication approach [8] tackles the aforementioned problems in the context of wide-area networks.

In the approach, multiple producer replicas send outputs to each consumer replica so that it can use whichever data arrives first. To further expedite processing, the approach allows replicas to independently process any available data, thereby causing multi-input replicas to produce outputs in different orders. Despite this complication, the approach always gives applications the results that would appear in the ideal non-replication scenario where the system is completely free from failures and delays. To achieve this notion of *replication transparency*, the approach merges disordered stream replicas, without blocking, into a non-duplicate stream at every input of an operator replica. For order-sensitive applications and operators (such as those with count-based windows), it sorts streams to ensure correctness. All the other operators are instrumented as non-blocking ones that always produce, from disordered streams, the tuples in the ideal non-replication scenario.

The approach also strives to achieve the best latency guarantee, relative to the cost of replication. For this, it first deploys replicas in a resource-efficient way and then garbage-collects and revives stream/operator replicas according to system conditions.

Passive Standby: Distributed Checkpointing and Parallel Recovery

Passive standby can flexibly trade off resource utilization for recovery speed by adjusting the checkpoint interval. Passive standby, however, 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 [9]. This approach groups servers into logical clusters. Inside each cluster, each server is collaboratively backed up by all other servers in the same cluster. This approach accelerates recovery because multiple servers cooperate to recover a failed one. During failure-free periods, it is also minimally intrusive to regular processing because it checkpoints only a few operators at a time and performs such tasks using idle CPU cycles.

KEY APPLICATIONS

- An important subset of stream processing applications involves critical, online monitoring tasks that require 24x7 operation. For example, IT administrators often want to monitor their networks for intrusions. Web site owners want to analyze and monitor click-streams to improve targeted advertising and to identify malicious users.

Brokerage firms want to analyze quotes from various exchanges in search for arbitrage opportunities. Phone companies want to process call-records for correct billing. These applications, and more, require the techniques described in this article.

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, 9, 18, 19] 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, Uğur Çetintemel, and Stan Zdonik. Fast and highly-available stream processing over wide area networks. In *Proc. of the 24th International Conference on Data Engineering (ICDE)*, April 2008.

[9] 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.

[10] 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.

[11] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[12] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[13] Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. In *STOC '97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 569–578, New York, NY, USA, 1997. ACM Press.

[14] 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.

[15] Fred B. Schneider. Byzantine generals in action: implementing fail-stop processors. *ACM Transactions on Computer Systems*, 2(2):145–154, 1984.

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

[17] 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.

[18] 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.

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

[20] 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.