Fast and Reliable Stream Processing over Wide Area Networks*

Jeong-Hyon Hwang, Uğur Çetintemel, and Stan Zdonik
Department of Computer Science
Box 1910, Brown University
Providence, RI 02912, USA
Tel: +1-401-863-6192
{jhhwang, ugur, sbz}@cs.brown.edu

Abstract

We present a replication-based approach that enables both fast and reliable stream processing over wide area networks. Our approach replicates stream processing operators in a manner where operator replicas compete with each other to make the earliest impact. Therefore, any processing downstream from such replicas can proceed by relying on the fastest replica without being held back by slow or failed ones. Furthermore, our approach allows replicas to produce output in different orders so as to avoid the cost of forcing an identical execution across replicas, without sacrificing correctness.

We first consider semantic issues for correct replicated stream processing and, based on a formal foundation, extend common stream-processing primitives. Next, we discuss strategies for deploying replicas. Finally, we present preliminary results obtained from experiments on PlanetLab that substantiate the potential benefits of our approach.

1 Introduction

Recently, there has been significant interest in applications where data, continuously generated at various locations in the world, needs to be processed in a timely fashion. Such *stream-processing* applications include financial market monitoring, asset tracking, seismic activity monitoring, call analysis, network monitoring and intrusion detection. In this application domain, low-latency processing is critical as it enables swift reaction to real-world events.

Stream processing systems are a class of software systems that efficiently facilitate implementation of stream processing applications [6, 4, 8]. In these systems, processing is typically expressed as an acyclic graph of operators that transform the data streaming through them. Many such systems are also geared toward distributed processing because a large body of applications inherently involve geographically dispersed data sources and engaging multiple servers can achieve highly scalable and available processing [9, 16, 20].

In this paper, we assume wide area networks as the substrate for extreme-scale stream processing. In such environments, it is possible to exploit a large number of computational elements to realize macroscopic applications that span diverse areas of the globe. Despite such benefits, the following reasons make it difficult to consistently guarantee *correct* and *timely* processing:

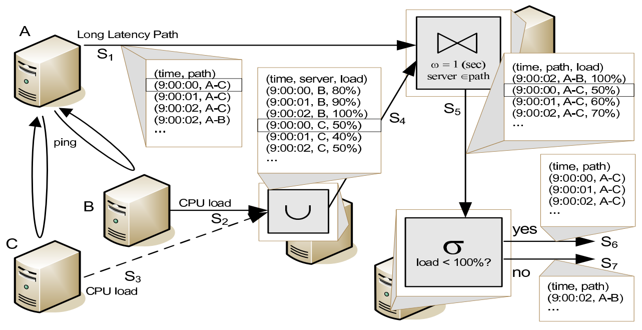
1. Servers are prone to failures. A server failure has a serious negative impact because a failed server blocks data flow and also may lose some state essential to processing. As processing involves more servers, the odds of experiencing failure increases accordingly.
2. Today's networks are vulnerable to link failure, link congestion, and malicious attacks. Communication outages sometimes last tens of minutes or more [7, 14].
3. Stream data often exhibits unexpected behaviors such as data surges [18, 20]. If a server cannot keep up with the incoming data, all the processing by subsequent servers will also be delayed.

To alleviate the limitations stated above, we adopt *operator replication* as the key mechanism. Specifically, we allow each operator in the system to receive inputs through *multiple replicated data flows* so that it can produce outputs relying on the *fastest input flow* and avoid being hindered by slow or faulty ones.

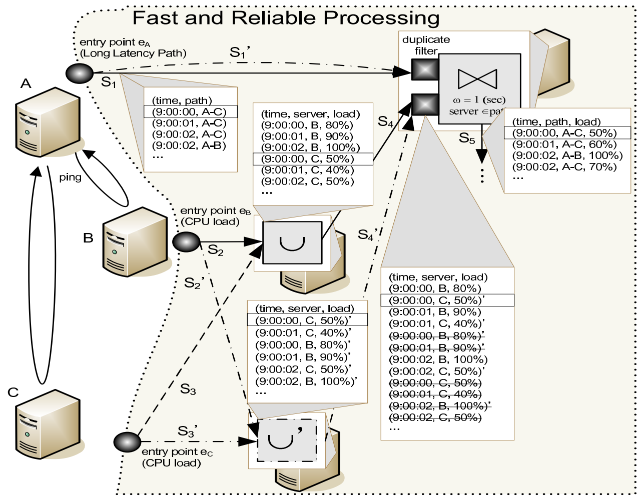
1.1 Motivating Example

Query. Figure 1(a) depicts a scenario where we are interested in finding long-latency communication paths among a subset of PlanetLab servers [2]. Those servers ping each

*This material is based upon work supported by the National Science Foundation under Grants No. IIS-0325838 and IIS-0086057. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.



(a) **Non-replicated Stream Processing**: In this example, tuple (9:00:00, A-C) arrives at $\bowtie$ on time through stream S_1. However, $\bowtie$ cannot immediately process that tuple because its matching tuple (9:00:00, C, 50%) arrives late as it was delayed on the way through stream S_3.



(b) **Replicated Stream Processing** (Processing after stream S_5 is omitted) : Replicating the input flows of $\bowtie$ (see that data flows S'_1, $S'_2 - S'_4$, $S'_3 - S'_4$ and operator U' are added) makes $\bowtie$ run in a more timely fashion. Replicated input flows however introduce duplicate tuples. In order to produce correct results, $\bowtie$ has to filter them out (see those stroked-through).

Figure 1. Example of Categorizing Slow Paths

other every second. If a server detects another server that does not respond in a second, the former reports this in the form of a data stream. In Figure 1(a), server A reports that communication path A-C has been slow or down since 9:00:00 and that also A-B became so at 9:00:02. However, such a delay might have appeared because the remote server slowly reacted to the ping message due to performing other important tasks. Therefore, we want to identify latencies resulting from network problems only (i.e., those certainly not caused by busy remote servers). In the example, the join operator $\bowtie$ correlates the report of slow paths with the report of remote servers' load readings, based on timestamp as well as server ID. The subsequent filter σ categorizes slow paths according to the load readings.

Impact of Slow Streams. As illustrated in Figure 1(a), a slow stream delays all processing downstream from it and may even delay the processing of other timely streams. In the example, the processing of a timely tuple (9:00:00, A-C) is delayed at $\bowtie$ as its matching tuple (9:00:00, C, 50%) arrives late.

Benefits of Replication. $\bowtie$ in the example is a stateful operator (i.e., the production of an output tuple usually involves multiple input tuples). For this reason, the timeliness of its operation is determined by the slowest input flow. As Figure 1(b) illustrates, replicating input flows allows the operator to obtain input tuples at earlier times (through the fastest among the replicated input flows) than in the non-replicated case. In principle, replication can improve both the *performance* (in terms of end-to-end latency) and *reliability* (in terms of the probability that the system satisfactorily performs the intended operation despite changes in the environmental conditions).

1.2 Problem Statements

Although replication has been extensively used in many areas, applying it in our framework raises the following research challenges.

1. What kind of *execution semantics* should replicated stream processing support?

   We take the position that replication must be *transparent* to users. Therefore, we aim at guaranteeing that the results generated from replicated stream processing, whether or not a failure happens, correspond to what non-replicated processing would produce. In Section 3, we discuss the details of this semantic guarantee.

2. How should we *extend processing primitives* for replicated stream processing?

   An operator with replicated inputs must filter out duplicate tuples as $\bowtie$ does in Figure 1(b). This is because processing such duplicates wastes processing/communication resources and also causes duplicate-sensitive operators (e.g., sum) to produce incorrect results. *Duplicate filtering* must be able to deal with cases where replicated streams produce different orders (compare streams S_4 and S'_4 in Figure 1(b)) and/or a data stream contains multiple identical tuples (refer to Section 3.5 for details). Furthermore, each operator should be implemented in a manner that expedites data flows (i.e., should not unnecessarily block data flows). We revisit this issue in Section 3.3.

3. How should we deploy replicas?

   Replication can improve both the *performance* and *reliability* of the system at the expense of using more *computation* and *network* resources. The key idea underlying our replica deployment approach is to replicate operators in a manner that most improves the performance and reliability, relative to the additional cost due to replication. Furthermore, while striving to avoid overload, our algorithm considers the relative importance of operators (e.g., how many end-users are dependent on them).

1.3 Road map

The rest of this paper is organized as follows. In Section 2, we give an overview of the topic and point out the contributions of this paper. Then, we devise a semantic model for correct replicated stream processing and present extended versions of stream-processing primitives that conform to the model (Section 3). In Section 4, we describe our algorithm for replica deployment. We present preliminary experimental results in Section 5. Finally, we conclude in Section 6.

2 Background

In stream processing, a query is expressed as a directed acyclic graph of operators that define how to transform the stream data [6, 4, 8]. Some stream operators are directly borrowed from relational algebra (e.g., Filter, Map, Union) and others are adapted (e.g., Aggregate, Join) [3]. The latter execute based on *windows* over data streams to cope with the infinite nature of streams (Refer to Section 3.3 for details). As Figure 1(a) shows, operators can be distributed over multiple servers for scalable processing [9, 16, 21].

2.1 Previous Work for Reliable Stream Processing

Most high-availability techniques that have been developed for stream processing [15, 12, 5, 11] assume a clear distinction between *"primary"* and *"backup (or secondary)"* processes. In such techniques, each primary executes queries and if it fails, a pre-dedicated backup takes over the failed execution.

Our previous work in [12] presents a broad classification of high-availability techniques depending on the way of maintaining backups: (1) In *passive standby*, each primary periodically checkpoints (i.e., copies *only the change* that occurred in its state since the last checkpoint) onto its backup. (2) *Active standby* uses *redundant execution* where each backup also receives and processes input data in parallel with its primary. (3) In *upstream backup*, each primary logs its output data. If a primary fails, upstream primaries flow their logged output data towards the backup of the failed one so that the backup can rebuild from scratch the failed process's state.

In our recent work [11], we use passive standby as the backup model for reliable stream processing in *local area clusters*. This is because that model has a distinct advantage of withstanding *high load* situations while gracefully degrading the recovery speed. Our approach in [11] partitions the query at each server into smaller pieces and backs them up onto different servers with different checkpoint frequencies so as to maximize the recovery speed. The approaches in [15, 5] fall into the active standby model. This model guarantees faster recovery than passive standby in *low load* situations where all the backups can also use the same amount of resources as the primaries. The upstream backup model incurs very low overhead during failure-free periods because the backups remain idle. However, it takes a long time to recover large state queries (e.g., recovering an aggregate with a window size of 10 minutes requires reprocessing 10 minute worth of tuples).

Recently, Murty and Welsh presented a high-level vision of a dependable architecture for Internet-scale sensing [13]. They proposed a replication technique that allows replicas to arbitrarily diverge and then reconciles results from such replicas by finding a representative value (such as the median).

2.2 Contributions

In this paper, we assume a wide area network where some sufficient amount of computation and communication resources are assigned for stream processing. Given these resources, we strive to replicate stream processing in order to improve both the performance and reliability of the system. Compared to the previous primary/backup techniques, our replication-based technique has the following advantages:

1. **High Performance**: In previous primary/backup approaches, only the primaries contribute to downstream processing. Therefore, adding more backups cannot improve the system performance. In our approach, replicas compete with each other to produce results early.

2. **No Failover Latency:** Previous primary/backup approaches recover from a failure by failover: (a) a backup ensures the problem of its primary after waiting for a certain amount of time, (b) sets up output streams to downstream processes on behalf of the failed one, and (c) starts sending output tuples downstream. It should be noticed that this failover procedure blocks data flows until it finishes. By contrast, our replication approach does not introduce any failover latency.

3. **No Blockage**: Previous approaches provide *precise recovery* (i.e., guarantee that the output with failover is the same as that from a failure-free execution) [12] in two different ways: (a) Make each primary postpone its output until its backup obtains all the information necessary to reproduce the output in the exactly same order (refer to "*determinants*" in [12] and "*output commit*" in [10]); (b) Enforce each operator to produce outputs *deterministically* (including the order) on both the primary and backup [5] (for example, each

Union replica postpones passing input tuple t until it ensures that all the upcoming tuples will have more recent timestamps than t). In principle, these methods block the flow of each tuple until it is guaranteed that the tuple will never violate their own invariants for precise recovery. Our approach does not introduce such blockage in processing.

In summary, this paper makes the following contributions:

1. We introduce a novel replication-based framework for fast and robust stream processing. This framework has the advantages stated above.
2. We define a semantic criterion for correct replicated stream processing. Based on this, we redesign processing primitives.
3. We discuss the problem of deploying replicas.
4. We present preliminary experimental results obtained from a prototype running on PlanetLab machines [2].

3 Replication Model

3.1 Assumptions

In this paper, we assume a wide area network with a sufficient amount of computation and communication resources for stream processing. We assume that the network layer runs a reliable, in-order, point-to-point message delivery protocol such as TCP. We take into account the fail-stop type of server/network failures and does not consider Byzantine failures where computers and networks behave in arbitrarily erroneous ways when there happen hardware failures, network congestion and disconnection, malicious attacks, etc.

We also assume that external stream sources connect to *entry points* that are scattered all over the world. Each entry point forwards input tuples from external sources to the downstream replicas, while buffering the tuples until all the downstream replicas receive them. In Figure 1(b), entry point e_2 forwards tuples from B to replicas $\cup$ and $\cup'$. To prepare for failure, each entry point also forwards input tuples to a number of peer entry points. If an external source loses connection to its current entry point, it redirects its output to a peer of the entry point. This new destination entry point then forwards input tuples to its peer entry points as well as the downstream replicas. If an entry point cannot access a downstream replica for a long time (while buffering input tuples), it may close its output to the replica to save its buffer space.

Finally, we assume that data sources timestamp tuples if they are equipped with well-synchronized clocks [1]. Otherwise, we assume that entry points timestamp tuples as they arrive from external sources. In this paper, we consider five different types of stream-processing operators, namely Filter, Map, Union, Aggregate, and Join.

3.2 Concepts

We take the position that replicas of the same query must produce *the same collection of tuples* from the same input. This is because different collections of tuples can be interpreted differently while suggesting conflicting actions in the real world. However, we allow individual replicas to produce streams in *different orders*. As discussed in Section 3.3, this relaxation allows multi-way operators, such as Union and Join, to output tuples immediately, without holding them to ensure an identical output order across replicas. This expedites all the related downstream processing.

Although replicas can produce streams in different orders, as illustrated in Section 3.5, we can merge such streams in a non-blocking manner, while eliminating duplicate tuples. Each stream obtained in this way has the same collection of tuples as the source streams. However, its order may be different from any one of the source streams. In Figure 1(b), the stream merged from S_4 and S_4' has a different order than S_4 as well as S_4'. This shows that our style of replication and merging strives to deliver outputs as early as possible, while introducing disorder in data streams.

In Section 3.3, we describe how we deal with streams disordered through replication. First, we introduce punctuations in data streams [19, 5]. In this paper, a punctuation is represented as a timestamp value with a guarantee that no more tuples will have a lower timestamp. Using punctuations, we extend stream-processing operators so that they can produce, from disordered input streams, the output tuples that non-replicated processing would produce. This guarantees that both the replicated processing and non-replicated processing produce the same collection of tuples, but possibly in different orders. In particular, as most of our operators are non-blocking, they usually produce disordered output streams from disordered input streams. As discussed later, the only exception is the operators with count-based windows. These operators run in a blocking manner to enforce an identical ordering across replicas.

The arguments above imply that if punctuations are managed properly, our replication model can deal with order difference in data streams. If an application at the output is order-sensitive, we can also sort the streams to the application based on the timestamps of tuples and punctuations. For this reason, we say that data streams are *consistent* with each other if they have the same contents (i.e., the same collection of tuples), despite any difference in their orders. We discuss the details of managing punctuations in Section 3.4.

Conceptually, consistent streams are permutations of the same collection of tuples. Definitions 1 and 2 formally express this notion of consistency.

Definition 1 ***(Contents of Stream)*** *Given a finite stream S, we define the contents of the stream as* $\mathcal{C}(S) :=$

$\{(t, \text{count}(t; S)) : t \in S\}$, *where* $\text{count}(t; S) := |\{k : S[k] = t\}|$ *is the number of occurrences of tuple* t *in stream* S *and* $S[k]$ *is the* k*th tuple in* S.

Notice that the contents of a stream correspond to the notion of multiset (or bag) in mathematics [17]. We use this notion to deal with cases where a stream contains multiple identical tuples. Notice that a stream with timestamps at too coarse granularity may contain identical tuples. A projection of tuples may also yield identical tuples.

Definition 2 ***(Consistent Streams)*** *Given two finite streams* S *and* S'*, we say that* S *and* S' *are "consistent" if* $\mathcal{C}(S) = \mathcal{C}(S')$ *(i.e., if they have the same contents). We also express this as* $S \equiv S'$.

Finally, we say that query Q guarantees "*replica consistency*" if its *replicas* are guaranteed to produce *consistent* output streams when they completely process *consistent* input streams. Practically speaking, this means that what a duplicate filter produces from parallel replicas of Q will also be consistent with what a single instance of Q would produce.

Definition 3 ***(Replica Consistency)*** *Let* Q *denote a query with* n *inputs and one output. Given a replica* R *of* Q*, let* $R(S_1, S_2, \cdots, S_n)$ *denote the output stream that* R *produces when it completely processes finite input streams* $S_1, S_2, \cdots, S_n$*. We say that* Q *guarantees replica consistency if it is always that* $R(S_1, S_2, \cdots, S_n) \equiv R'(S'_1, S'_2, \cdots, S'_n)$ *for any replicas* R *and* R' *of* Q *and for any finite input streams* $\{S_i\}_{i=1}^n$ *and* $\{S'_i\}_{i=1}^n$ *such that* $S_i \equiv S'_i$ $(1 \leq i \leq n)$.

3.3 Operators

In this section, we discuss how we redesign stream-processing operators for replica consistency.

Theorem 1 shows that a query must guarantee replica consistency if all its constituting operators guarantee replica consistency.

Theorem 1 *Let* Q *denote a query with* n *inputs and one output. If all the constituent operators of* Q *guarantee replica consistency,* Q *also guarantees replica consistency (i.e.,* $R(S_1, S_2, \cdots, S_n) \equiv R'(S'_1, S'_2, \cdots, S'_n)$ *for any replicas* R *and* R' *of* Q *and any input streams* $\{S_i\}_{i=1}^n$ *and* $\{S'_i\}_{i=1}^n$ *such that* $S_i \equiv S'_i$*).*

Proof: If Q contains only one operator, trivially Q guarantees replica consistency. Otherwise, denote the terminal operator of Q as o (i.e., o produces the output stream of Q). Then, for each input j $(1 \leq j \leq m)$ of operator o, let Q_j be the query that consists of operators upstream from j. For a replica R of Q, let $R_j(S_1, S_2, \cdots, S_n)$ denote the sequence of tuples that have appeared on j until R completely processes input streams $S_1, S_2, \cdots, S_n$. Since each Q_j has less operators than Q (note that $o \notin Q_j \subseteq Q$), by the induction hypothesis, we get that $R_j(S_1, S_2, \cdots, S_n) \equiv R'_j(S'_1, S'_2, \cdots, S'_n)$ for another replica R' of Q and input streams $\{S_i\}_{i=1}^n$ and $\{S'_i\}_{i=1}^n$ such that $S_i \equiv S'_i$ $(1 \leq i \leq n)$. Since o also guarantees replica consistency, the replicas of o in R and R' produce consistent output streams as they receive consistent input streams $\{R_j(S_1, S_2, \cdots, S_n)\}_{j=1}^m$ and $\{R'_j(S'_1, S'_2, \cdots, S'_n)\}_{j=1}^m$, respectively. As a result, we get that $R(S_1, S_2, \cdots, S_n) \equiv R'(S'_1, S'_2, \cdots, S'_n)$. □

Theorem 1 can be generalized for a query with multiple outputs by forming sub-queries consisting of operators upstream from each output and applying the theorem to such sub-queries. Hereafter, we revisit stream-processing operators and extend them (if necessary) for replica consistency.

3.3.1 Stateless Operators

Stateless operators are those that produce each output tuple based on only the most recent input tuple. Filter, Map, Union belong to this category. Filter outputs a tuple from its input stream if the tuple satisfies a pre-defined predicate. Map converts each input tuple into an output tuple of a different format. Union merges two or more streams into a single output stream.

It should be noted that stateless operators guarantee *replica consistency* without extension. This is because they always make a *deterministic* decision (i.e., replicas are guaranteed to make the same decision) for each input tuple regardless of the orders (and also the inter-arrival orders) of input tuples. For example, Union each time passes the exact input tuple that it received. Notice that Filter and Map are *deterministic* because they always produce a unique output stream from an input stream. On the contrary, our implementation of Union can produce different output streams (in terms of order) from the same set of input streams as it immediately outputs whichever input tuple comes first. In this sense, Union is *non-blocking* and also *nondeterministic*. Union can introduce *disorder* into its output stream even when its input streams are ordered.

3.3.2 Join

Join has two inputs I_1 and I_2. Whenever a tuple appears on one input, Join searches for *matching* tuples among those that have appeared on the other input (for each input I_i, Join has a buffer B_i to remember past tuples). We say that a pair of input tuples t_1 and t_2 (from I_1 and I_2, respectively) *match* if (a) they belong to the same time window w (i.e., $|t_1.\text{timestamp} - t_2.\text{timestamp}| < w$) and (b) they satisfy predicate P associated with the operator (i.e., $P(t_1, t_2)$ holds). When this operator finds such matching input tuples t_1 and t_2, it *immediately* produces the concatenation $t_1 \otimes t_2$ of them as an output tuple (in this sense, our implementation of Join is *non-blocking*). This Join algorithm is summarized in Figure 2.

```
When $\bowtie$ receives input tuple $t_1$ via input $I_1$
(Input tuples via $I_2$ are processed symmetrically; lines 06-08 are
 for garbage collection)
  01.  For each $t_2 \in B_2$ such that
  02.      $|t_1.\text{timestamp} - t_2.\text{timestamp}| < w$ and $P(t_1, t_2)$,
  04.          output $t_1 \otimes t_2$;
  05.  $B_1 \leftarrow B_1 \cup \{t_1\}$;
  06.  $I_1.\mathcal{S}[t_1.\text{source\_id}] \leftarrow t_1.\text{punctuation}$;
  07.  $I_1.\text{punctuation} \leftarrow I_1.\mathcal{S}.\text{get\_min\_punctuation}()$;
  08.  $B_2 \leftarrow B_2 - \{t_2 \in B_2 :$
                                  $t_2.\text{timestamp} < I_1.\text{punctuation} - w\}$;
```

Figure 2. Join Algorithm

In this paper, we assume that Join sets the timestamp of each output tuple to the most recent timestamp of the matching input tuples. This is to compute the end-to-end latency of each output tuple based on its timestamp. Notice that, in an ideal setting where all the processing and transmission tasks do not cause any latency, each output tuple would be generated as soon as the last related input tuple is produced. The timestamp of each output tuple captures this ideally earliest generation time of the tuple.

It is easy to see that Join guarantees *replica consistency* because, for each pair of matching input tuples, each replica exactly once produces the concatenation of those tuples regardless of the inter-arrival order of the input streams. Since each output tuple is produced only when the last contributing tuple arrives, the pace of Join is very sensitive to the *slowest input*. In Section 5, we experimentally demonstrate this point. It should be also noticed that Join, similarly to Union, is *nondeterministic* and also can introduce *disorder*.

One question that arises at this point is how to determine and remove (to save space) input tuples that will no longer be used. We need to solve this *garbage collection* problem even when the input streams are disordered by Unions and Joins upstream. We solve this problem by annotating each tuple on input I_i with $(\text{source\_id}, \text{punctuation})$. This pair guarantees that input I_i will never receive tuples that (1) originated from stream source source_id and (2) have timestamps earlier than punctuation. In other words, this means that I_i has already seen the *entirety* of the tuples with origin at source_id and timestamps upto $\text{punctuation} - \epsilon$, where ϵ is the unit of timetamp. In Section 3.4, we discuss this notion of *bounded entirety* in detail. For garbage collection, we also assume that input I_i knows the set $I_i.\mathcal{S}$ of stream sources that are upstream from I_i. Based on this, we can find the smallest punctuation across the stream sources in $I_i.\mathcal{S}$ (see $I_1.\text{punctuation}$ in line 07 of Figure 2). Because none of the tuples incoming via I_i will have timestamps earlier than $I_i.\text{punctuation}$, we can then find (and discard) the tuples on the other input that will never be used for processing (see line 08 in Figure 2). Finally, it should be noted that steps in lines 06 and 07 can be done with a low cost if we implement $I_i.\mathcal{S}$ as a heap that places the smallest punctuation value at the root. In this case, the complexities of the steps are $\log(|I_i.\mathcal{S}|)$ and $\log(1)$, respectively.

```
When receives input tuple $t$ belonging to group $g$
  01.  for each $w \in \mathcal{W}(t.\text{timestamp})$
  02.      $g.\text{windows}[w].\text{update}(t)$;
  03.  $g.\mathcal{S}[t_1.\text{source\_id}] \leftarrow t_1.\text{punctuation}$;
  04.  $g.\text{punctuation} \leftarrow g.\mathcal{S}.\text{get\_min\_punctuation}()$;
  05.  for each $\overline{w} \in g.\text{windows}$ such that $\overline{w} < g.\text{punctuation}$;
  06.      output $\overline{w}.\text{get\_summary}()$;
  07.      $g.\text{windows} \leftarrow g.\text{windows} - \{\overline{w}\}$;
```

Figure 3. Agg. with Time-based Windows

3.3.3 Aggregate

Aggregate splits input stream I into substreams $\{I[g] : g \in \mathcal{G}\}$ where $\mathcal{G}$ is the set of all groups and $I[g]$ is a subsequence of I that contains tuples belonging to group g. For each substream $I[g]$, this operator assumes windows (sets of tuples) that are typically defined based on either *timestamps* or *count of tuples*. If a window expires, Aggregate produces an output tuple computed from the tuples that belonged to the window. It is easy to see that Aggregate will guarantee *replica consistency* if it deterministically forms and closes windows (i.e., if the corresponding windows across replicas are guaranteed to obtain the same collection of tuples until they expire). In this case, Aggregate will also become *deterministic* as it will always produce a unique output stream from the same input stream.

Aggregate with Time-Based Windows. In this case, each group g forms windows of w seconds every s seconds. Therefore, for each input tuple t, g can determine the set of windows $\mathcal{W}(t.\text{timestamp})$ that t belongs to. For example, when $w = 10$ (sec) and $s = 5$ (sec), for a tuple with timestamp of 9:00:43, we get $\mathcal{W}(\text{9:00:43}) = \{[\text{9:00:35}, \text{9:00:45}), [\text{9:00:40}, \text{9:00:50})\}$ (see line 01 in Figure 3). Then, for each window found above, g updates (based on tuple t) the aggregate value that the window has maintained so far (see line 02 in Figure 3). For replica consistency, we let each window $\overline{w}$ expire only when it is certain that $\overline{w}$ will no longer see a new tuple. As we did for Join, we assume that each group g knows the set $g.\mathcal{S}$ of stream sources that produce tuples belonging to g. Given this knowledge, group g can find its current punctuation (line 04) and then safely closes any window $\overline{w}$ whose expiration time precedes the punctuation (lines 05-07). The timestamp of each output tuple is set to the expiration time of the window because this is the ideally earliest generation time of that tuple.

Aggregate with Count-Based Windows. This operator maintains, for each group g, a window of w tuples that skips every s tuples each time it moves. As the execution of this operator is sensitive to the order of input tuples, this operator sorts the input tuples, before processing them, according to a *deterministic* ordering rule (e.g., for deterministic ordering, this rule may break tie in timestamp by source ID). Notice that this is to reconstruct a sequence of input tuples

```
When receives input tuple $t$ belonging to group $g$
01. $g.B \leftarrow g.B \cup \{t\}$;
02. find $g$.punctuation as in lines 03 and 04 of Figure 3;
03. $L \leftarrow \text{sort}(\{t \in g.B : \text{t.timestamp} < \text{g.punctuation}\})$
04. while $\exists(t \leftarrow L.\text{front}())$
05.     $\text{count} \leftarrow \text{count} + 1$;
06.     for each $w \in \mathcal{W}(\text{count})$
07.     $g.\text{windows}[w].\text{update}(t)$;
08.     for $\overline{w} \in g$.windows such that $\overline{w} < \text{count}$;
09.         output $\overline{w}.\text{get\_summary}()$;
10.         $g.\text{windows} \leftarrow g.\text{windows} - \{\overline{w}\}$;
```

Figure 4. Agg. with Count-based Windows

that would appear if none of the upstream components introduce latency. Figure 4 shows the detailed algorithm for this operator.

3.4 Bounded Entirety

As described in Section 3.3, our stream-processing operators realize replica consistency as well as garbage collection, based on $(\text{source\_id}, \text{punctuation})$ annotated to each tuple. Conceptually, such annotation turns a stream S into sub-streams with a guarantee that the entirety of sub-stream $S[\text{source\_id}]$ is seen up to timestamp $(\text{punctuation} - \epsilon)$. In this section, we discuss how we can preserve such a guarantee of *bounded entirety* at every operator in the query.

We first assume that tuples are initially timestamped in the ascending order at data sources (or at entry points). In this case, source_id of each tuple t is set to the identifier of the source and $t.\text{punctuation}$ is set to $t.\text{timestamp}$. Notice that streams formed in that way provide the guarantee of bounded entirety. Hereafter, we describe how each operator relays this guarantee from its input streams to output stream.

Filter & Map. These operators are FIFO (i.e., order-preserving) unary operators. Thus, we set the punctuation of each output tuple to the punctuation of the input tuple

Union. If inputs of this operator are commonly downstream from the same stream source, the punctuation for a tuple is set to the minimum among the most recent punctuations, one for each input of the operator. Otherwise, we leave the punctuation of each tuple as it is.

Join. Because each output tuple t is contributed by two input tuples t_1 and t_2 (i.e., two source_id are involved), we introduce a new source_id for each pair of source_ids. In other words, given such a mapping μ, we set $t.\text{source\_id}$ to $\mu(t_1.\text{source\_id}, t_2.\text{source\_id})$. Notice that tuples of origin at either $t_1.\text{source\_id}$ or $t_2.\text{source\_id}$ will never have timestamps earlier than both $t_1.\text{punctuation}$ and $t_2.\text{punctuation}$. Therefore, we set $t.\text{punctuation}$ to $\min(t_1.\text{punctuation}, t_2.\text{punctuation})$.

Aggregate. Output tuples of the same group are produced in the ascending order of timestamps. Therefore, we assign a new source_id for each group and set $t.\text{punctuation}$ of each output tuple t to $t.\text{timestamp}$.

```
When receives tuple $t$ through a replicated stream $S_i$
01. $\mathfrak{R}[t.\text{source\_id}][i] \leftarrow t.\text{punctuation}$;
02. $w \leftarrow \mathfrak{R}[t.\text{source\_id}].\text{get\_max\_punctuation}()$;
03. if $t.\text{timestamp} \geq w$
04.     $\text{count}[t][i] \leftarrow \text{count}[t][i] + 1$;
05.     if $\text{count}[t][i] > \text{max\_count}[t]$
06.         pass $t$ to operator $o$;
07.         $\text{max\_count}[t] \leftarrow \text{count}[t][i]$;
```

Figure 5. Duplicate Filtering

3.5 Duplicate Filtering

As described in Section 1, our replication model allows each operator to obtain tuples from replicated input flows. For such an operator to run as it would do with non-replicated inputs, we need to filter out *duplicate tuples* as they arrive. Conceptually, duplicate tuples are those that arrive later than the corresponding tuples from other replicated streams.

We implement duplicate filters based on the fact that each replica is guaranteed to produce the same collection of output tuples. As pointed out in Section 1.2, a duplicate filter must be able to deal with different orders of replicated streams as well as multiple occurrences of identical tuples on data streams. Furthermore, it should not block data flow.

Figure 5 describes our algorithm for duplicate filtering. Complying with the multiset-based notion of stream contents (refer to Definition 1), we use a variable $\text{count}[t][i]$ to count how many times each tuple t appears on a replicated stream S_i (see line 04). If another replicated stream has already flowed t more often than S_i, we filter out t (i.e., do not pass t to the next the operator). Notice that the current tuple t in this case is considered as duplicate because it arrived later than some other (line 05). Lines 01-03 are to rule out duplicates at an early stage, without using count variables. They are based on the reasoning that if the punctuation of another replicated stream succeeds the timestamp of tuple t, that stream must have completed sending t (i.e., the current tuple t is duplicate). Such a punctuation can also be used to discard counters that will no longer be used afterwards (details are omitted in Figure 5). Theorem 2 proves the correctness of the duplicate filtering algorithm

Theorem 2 *Let $\mathfrak{D}(S_1, S_2, \cdots, S_n)$ denote the sequence of tuples that duplicate filter $\mathfrak{D}$ has output from streams $S_1, S_2, \cdots, S_n$. If those streams are consistent with each other, $\mathfrak{D}(S_1, S_2, \cdots, S_n)$ is also consistent with those streams (i.e., $\mathfrak{D}(S_1, S_2, \cdots, S_n) \equiv S_i$ for all $1 \leq i \leq n$).*

Proof: It is sufficient to prove that $\mathfrak{D}$ outputs each tuple t $\text{count}(t; S_i)$ times. Let $m := \text{count}(t; S_i)$. (1) if $m = 0$, none of the streams $\{S_i\}$ contains tuple t. Therefore, $\mathfrak{D}$ does not output t. (2) For $m > 0$, without loss of generality, let stream S_1 pass its mth instance of t to $\mathfrak{D}$ earlier than any other stream. Since $t.\text{punctuation} \leq t.\text{timestamp}$, we get

```
01. $\mathfrak{G}(Q) \leftarrow \{(q_1, q_2, \cdots, q_m) : q_i \subset Q,$
02.     $\mathrm{load}_i + \Delta\mathrm{load}_i(q_i) < \alpha C_i,$
03.     $\mathrm{latency}(S) < \theta$ for any stream $S$ related to $(q_i)_{i=1}^m$
04. Find $(q_1^*, q_2^*, \cdots, q_m^*) \in \mathfrak{G}(Q)$ such that
05.     $\frac{\mathrm{gain}(q_1^*, q_2^*, \cdots, q_m^*)}{\mathrm{cost}(q_1^*, q_2^*, \cdots, q_m^*)} \geq \frac{\mathrm{gain}(q_1, q_2, \cdots, q_m)}{\mathrm{cost}(q_1, q_2, \cdots, q_m)}, \forall (q_i) \in \mathfrak{G}(Q)$
06.     where $\mathrm{gain}(q_i) := \Delta U_{\mathrm{lat}}(q_i) \cdot \Delta U_{\mathrm{rel}}(q_i),$
07.         $\mathrm{cost}(q_i) := \frac{\sum_{i=1}^m \Delta\mathrm{load}_i(q_i)}{\sum_{i=1}^m (\alpha C_i - \mathrm{load}_i(q_i))} \cdot \frac{\Delta B(q_i)}{\alpha \hat{B} - B}.$
```

Figure 6. Replica Deployment

that $\mathfrak{R}[t.\mathrm{source\_id}][1] \leq t.\mathrm{timestamp}$ (refer to line 01 in Figure 5). As other streams will flow t later, we also get that $\mathfrak{R}[t.\mathrm{source\_id}][i] \leq t.\mathrm{timestamp}$ for all $1 < i \leq n$. Thus, $w \leq t.\mathrm{timestamp}$ (line 02). Then, it is easy to see that $\mathfrak{D}$ will output t (lines 03-06). Also note that this is the mth output of t because, by the induction hypothesis, $\mathfrak{D}$ must have output t, $(m-1)$ times before. $\mathfrak{D}$ then sets both $\mathrm{count}[t][1]$ and $\mathrm{max\_count}[t]$ to m. It is easy to see that $\mathfrak{D}$ will filter out t afterwards because, for all $1 < i \leq n$, $\mathrm{count}[t][i]$ cannot be larger than $\mathrm{max\_count}[t] = m$.

4 Replica Deployment

In this paper, we assume that servers are grouped into logical clusters. We also assume that servers $\{\mathfrak{S}_i\}_{i=1}^m$ in a group $\mathfrak{G}$ elect a coordinator that keeps track of the CPU utilization as well as the network utilization of them. Hereafter, using the algorithm in Figure 6, we sketch our algorithm for replicating operators in query network Q. We denote a replica deployment plan with $(q_1, q_2, \cdots, q_m)$. Each $q_i \subset Q$ represents the operators that server $\mathfrak{S}_i$ will execute.

Our first criterion for replica deployment is that the deployment plan must be *feasible*. In other words, the replica must not overload any server in the system. We represent this condition as $\mathrm{load}_i + \Delta\mathrm{load}_i(q_i) < \alpha C_i$ (see line 02), where load_i is the current load of server $\mathfrak{S}_i$ and $\Delta\mathrm{load}_i(q_i)$ is the the expected amount of load that will increase when $\mathfrak{S}_i$ runs q_i. C_i represents the processing capacity of server $\mathfrak{S}_i$ and $\alpha < 1$ is a constant to have headroom in CPU availability. If a server becomes overloaded, the sever may abandon replicas selectively (e.g., those less important than others or those more replicated than others).

We also note that the replica to deploy must not be hindered by slow communication paths. In other words, for any data stream S related to the replica, the transmission latency $\mathrm{latency}(S)$ must be kept low (line 03). In Figure 6, $\mathfrak{G}(Q)$ represents the theoretical collection of all feasible deployment plans.

As pointed out in Section 1.2, replica deployment should be done carefully, while considering both the *gain* in terms of *performance* and *reliability*, as well as the *cost* of using more *network* and *computation* resources. We use end-to-end latency as the main metric for system performance. However, as this quantity usually has a long tailed distribution [14] (also refer to Figure 8), we use a representative value with cutoff (e.g., average under pth percentile). For a replica deployment plan $(q_1, q_2, \cdots, q_m)$, we first estimate how much latency the replica will reduce in terms of the metric described above. Then, we evaluate the gain in performance, while also considering the popularity of the replica (i.e., how many users will eventually benefit from the replica). In Figure 6, $\Delta U_{\mathrm{lat}}(q_i)$ represents this estimated gain in performance.

We also assume that users are tolerant to latencies only to a certain point (e.g., latencies of 10 seconds or more imply the failure of service, for whatever reasons they appear). Based on this, we define *reliability* as the *probability* that latencies remain under the threshold. We also estimate the benefit $\Delta U_{\mathrm{rel}}(q_i)$ in terms of reliability for the case of deploying replica $(q_1, q_2, \cdots, q_m)$.

Finally, we capture the cost of adding a replica by considering the increase in utilization for both the computation and network resources, relative to their current availability. In Figure 6, $\Delta\mathrm{load}_i(q_i)$ and $\Delta B(q_1, q_2, \cdots, c_m)$ denote the estimated increase in the processing load of server $\mathfrak{S}_i$ due to running q_i and the increase in network utilization due to the message traffic for replica $(q_1, q_2, \cdots, q_m)$, respectively. $\alpha C_i - \mathrm{load}_i(q_i)$ represents the current CPU availability of server $\mathfrak{S}_i$. $\alpha\hat{B} - B$ represents the current bandwidth availability of the network.

Discussion. As Figure 6 shows, our replica deployment algorithm chooses a deployment plan with the highest *"gain/cost"* ratio. We can see that this algorithm will prefer operators sensitive to slow input flows (e.g., Join) over other operators (e.g., Union) as the former will get more benefit from replicated input flows than the latter (refer to Figure 8 for the sensitivity of these operators to slow input flows). We can also see that the algorithm will favor popular operators. This also means that upstream operators are in general more preferred to downstream operators because they affect more end-users. We are currently looking for a heuristic that efficiently finds the set of operators that will lead to the highest benefit, relative to the overhead of replicating them.

5 Experimental Results

In this section, we present preliminary results obtained from experiments on PlanetLab machines [2]. For all the experiments that we conducted, we used the servers illustrated in Figure 7. In more detail, each server at MIT, UW, UCSD, and UFL instantiated two stream generators, one that reports the server's CPU utilization every half a second and the other that reports the ping latency from the server to other servers again every half a second. The load readings of those servers were sent to a Union $\cup_1$ at WISC while the latencies to remote servers were sent to another Union $\cup_2$ at Purdue. Then, the Join operator $\bowtie$ at OSU correlated the load readings and latencies, based on both timestamp and

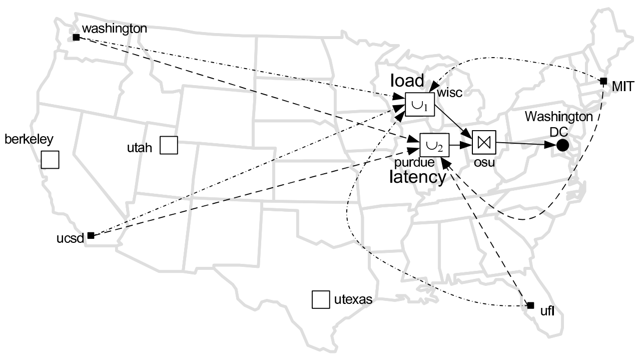


Figure 7. Experimental Setup

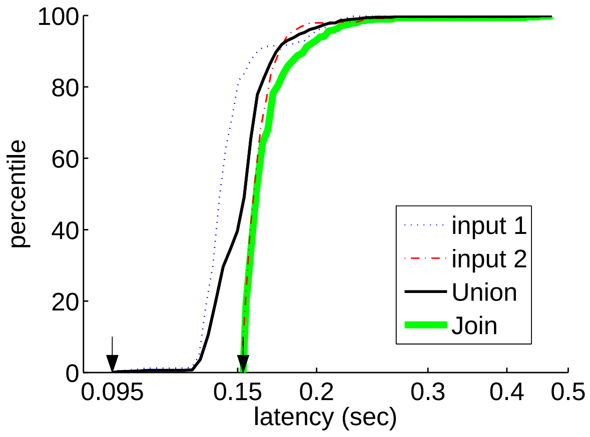


Figure 8. Sensitivity to Slow Input Streams

server ID. Notice that we set up the experiment similarly to the example in Section 1 except that input tuples were generated every half a second.

5.1 Sensitivity to Slow Inputs

In the first experiment, we study the sensitivity of stream processing operators to slow inputs. Figure 8 illustrates the results for two representative multi-way operators, namely Union and Join. In the experiment, we placed these operators at OSU and let both of them receive inputs from $\cup_1$ and $\cup_2$. In the figure, a point (x, y) on a curve represents that the latency of x seconds is the yth percentile, for the output tuples that have appeared for an hour on the specified stream. For example, the figure shows that the smallest latency at the output of Union was 95 (msec) whereas that at the output of Join was 150 (msec). It also shows that the median latencies of them, however, get closer to each other (notice that the 50th percentile means the median latency). In summary, we can see that the pace of Join is indeed determined by the slowest input (i.e., input 2 in the figure). This is because this operator cannot produce an output tuple until both the contributing input tuples arrive, one from

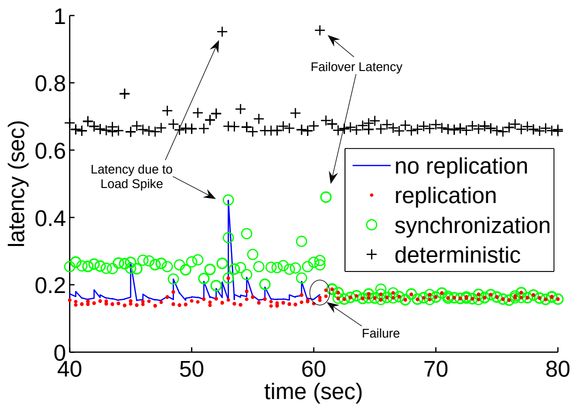


Figure 9. Latency under Different High-Availability Techniques

each input. By contrast, Union is less sensitive than Join in the sense that it immediately outputs whichever comes first. Note that replicating the data flow to input 2 may allow those operators output at earlier times than before. If that happens, Join will get more benefit than Union.

5.2 Comparison of Techniques for Reliable Stream Processing

In this section, we compare techniques proposed for reliable stream processing. In Figure 9, the curve for "no replication" represents how the latency varies over time for the tuples coming out of $\bowtie$ at OSU. In the experiment, we crashed the stream processing engine at WISC at time 60. Therefore, $\bowtie$ does not produce output tuples as it no longer receives input tuples from $\cup_1$. By contrast, other curves show that reliability techniques indeed protect stream processing from failure. The curve for "replication" shows that our replication technique, as expected, provides reliability against failure but also reduces latency. For this case, we set up a butterfly network where replicas $\cup_1'$ and $\cup_2'$ of $\cup_1$ and $\cup_2$, respectively, are located at Purdue and WISC. Notice that the latency increases after failure at time 60 as $\bowtie$ no longer can benefit from replication due to the loss of $\cup_1$ and $\cup_2'$. In Figure 9, "synchronization" corresponds to the case where primary $\cup_1$ and its backup $\cup_1'$ (and also $\cup_2$ and $\cup_2'$) enforce an identical ordering between them by use of determinants (refer to Section 2.2 and also [12]). In this case, each primary holds output tuples until it turns out that the related determinants have arrived at the backup, while in principle trading off reliability against latency. At time 60, the curve for this case drops down to that for "replication" because $\cup_1'$ and $\cup_2$ at Purdue do not have partners to synchronize with. Finally, "deterministic" shows the cost in latency incurred by the method in [5]. In this case, as described in Section 2.2, both primary $\cup_1$ and its backup $\cup_1'$

(and also primary $\cup_2$ and its backup $\cup_2'$) run deterministically (including the order) by holding tuples until they assure an identical ordering of those tuples. In the figure, the increase in latency is mainly due to the interval of generating input tuples at the remote data sources. For this reason, this extra latency will decrease as the data rate increases (i.e., the interval decreases). However, such a reduction in latency will eventually be bounded by the latency of the slowest input to the Unions.

In summary, our replication-based approach improves both *performance* as well as *reliability* when extra resources are available. This sharply contrasts with previous approaches that can only trade off reliability against performance. As illustrated in Figure 9, our approach does not introduce failover latency.

6 Conclusion

In this paper, we introduce a replication-based approach to achieve fast and reliable stream processing over wide area networks. Our technique provides replica transparency, guaranteeing that replicated processing, regardless of failures and congestions, will produce what non-replicated processing would produce if all the system components are free from failure and latency. In our approach, replicas compete with each other to make the earliest impact, while guaranteeing that they will eventually produce the same collection of tuples. These replicas, however, can run in different orders so as to avoid the overhead of other previous approaches. Our technique can merge replicated streams into a single stream, while filtering out duplicates. If required by the end application, it can also be instructed to restore the order of the stream that non-replicated processing would produce.

The contributions of this paper include: (1) a model for replica transparency, (2) re-designed stream processing operators that comply with the replication model, (3) a mechanism for duplicate filtering, (4) discussion on deploying replicas, and finally (5) preliminary experimental results.

We currently have a prototype system that enables manual allocation of stream-processing operators over PlanetLab machines. We intend to extend the system while implementing the replica placement strategy described in the paper. We also plan to conduct a comprehensive set of experiments to characterize the run-time behavior and performance of our approach.

References

[1] `http://www.ntp.org`.

[2] `http://www.planet-lab.org`.

[3] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A new model and architecture for data stream management. *The VLDB Journal*, Sep 2003.

[4] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proc. of 2002 ACM PODS*, June 2002.

[5] M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker. Fault-tolerance in the borealis distributed stream processing system. In *Proc. of the 2005 ACM SIGMOD*, June 2005.

[6] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams: A new class of data management applications. In *Proc. of the 28th VLDB*, Aug. 2002.

[7] B. Chandra, M. Dahlin, L. Gao, and A. Nayate. End-to-end wan service availability. In *In Proc. of the 3rd USITS, San Francisco, CA*, pages 97–108, 2001.

[8] S. Chandrasekaran, A. Deshpande, M. Franklin, and J. Hellerstein. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proc. of the 1st CIDR*, Jan. 2003.

[9] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, Y. Xing, and S. Zdonik. Scalable distributed stream processing. In *Proc. of the 1st CIDR*, 2003.

[10] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, 2002.

[11] J.-H. Hwang, , U. Çetintemel, and S. Zdonik. A cooperative, self-configuring high-availability solution for stream processing. In *Proc. of the 23th ICDE*, 2007.

[12] J.-H. Hwang, M. Balazinska, A. Rasin, U. Çetintemel, M. Stonebraker, and S. Zdonik. High-availability algorithms for distributed stream processing. In *Proc. of the 21th ICDE*, 2005.

[13] R. N. Murty and M. Welsh. Towards a dependable architecture for internet-scale sensing. In *2nd Workshop on Hot Topics in Dependability (HotDep'06),*, 2006.

[14] V. Paxon. End-to-end routing behavior in the internet. *IEEE ACM Transactions on Networking*, 5(5):601–615, 1997.

[15] M. A. Shah, J. M. Hellerstein, and E. Brewer. Highly-available, fault-tolerant, parallel dataflows. In *Proc. of the 2004 ACM SIGMOD*, June 2004.

[16] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *Proc. of the 19th ICDE*, Mar. 2003.

[17] R. P. Stanley. *Enumerative Combinatorics*. Cambridge University Press, 1997.

[18] N. Tatbul, U. Çetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker. Load shedding in a data stream manager. In *Proc. of the 29th VLDB*, Aug. 2003.

[19] P. Tucker, D. Maier, T. Shreard, and L. Fegaras. Exploiting punctuation semantics in continuous data streams. *IEEE TKDE*, 15(3):555–568, 2003.

[20] Y. Xing, J.-H. Hwang, U. Çetintemel, and S. Zdonik. Providing resiliency to load variations in distributed stream processing. In *Proc. of the 32th VLDB*, Sept. 2006.

[21] Y. Xing, S. Zdonik, and J.-H. Hwang. Dynamic load distribution in the borealis stream processor. In *Proc. of the 21th ICDE*, Mar. 2005.