Fast and Highly-Available Stream Processing over Wide Area Networks

Jeong-Hyon Hwang, Uğur Çetintemel, and Stan Zdonik
Department of Computer Science, Brown University
{jhhwang, ugur, sbz}@cs.brown.edu

***Abstract*— We present a replication-based approach that realizes both fast and highly-available stream processing over wide area networks. In our approach, multiple operator replicas send outputs to each downstream replica so that it can use whichever data arrives first. To further expedite the data flow, replicas run independently, possibly processing data in different orders. Despite this complication, our approach always delivers what non-replicated processing would produce without failures. We call this guarantee replication transparency.**

In this paper, we first discuss semantic issues for replication transparency and extend stream-processing primitives accordingly. Next, we develop an algorithm that manages replicas at geographically dispersed servers. This algorithm strives to achieve the best latency guarantee, relative to the cost of replication. Finally, we substantiate the utility of our work through experiments on PlanetLab as well as simulations based on real network traces.

I. Introduction

Recently, there has been significant interest in applications where high-volume data streams need to be processed with low latency. Such applications include network monitoring and intrusion detection, seismic activity monitoring, Web feed analysis, global asset tracking, and monitoring of large ecosystems. 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 facilitate implementation of stream processing applications [1], [2], [3]. In these systems, processing is typically expressed as an acyclic graph of operators that transform the data streaming through them. These systems are usually geared toward distributed processing because many applications inherently involve geographically dispersed data sources and a good use of multiple servers can achieve highly scalable processing [4], [5], [6].

In this paper, we consider stream processing in a macro-scale that spans diverse areas of the globe. This will allow us to monitor various events occurring around the world and make smart decisions in near real time. To realize correct and timely processing, however, we must address the following challenges:

1) As we use more servers, server failures are more likely to occur. A failed server cannot send data and may have lost data essential to processing.
2) Computer networks are vulnerable to link failures and congestion. Communication outages sometimes last tens of minutes or more [7], [8].
3) A server can be overloaded due to unexpected surges of data streams [6] or by other applications that share the server. In this case, stream processing at subsequent servers also gets delayed.

This work has been supported by the National Science Foundation under Grant No. IIS-0086057 and IIS-0325838.

We observe that previous techniques for reliable stream processing [9], [10], [11], [12] cannot successfully address the challenges above. These techniques commonly deploy, for each operator, k replicas on independent servers to tolerate up to (k-1) simultaneous failures. In these techniques, however, only one of the peer replicas can feed a downstream replica. If such a replica fails (or gets overloaded/disconnected), the subsequent processing stalls until the downstream replica notices the problem and acquires a new input connection from another functioning upstream replica. Furthermore, these previous techniques run replicas identically at distant servers, thereby introducing extra delays.

To overcome the limitations of previous techniques, we propose a new approach where multiple replicas send outputs to each downstream replica so that it can use whichever data arrives first. To further expedite the data flow, our approach also allows replicas to independently process any available data. This may cause multi-input replicas to produce outputs in different orders. Despite such complications, our approach always delivers the results that non-replicated stream processing would produce without failures. We call this guarantee *replication transparency*.

Our approach uses more resources than previous approaches because all replicas send outputs downstream. However, our approach has a distinct advantage of improving performance since it always uses the fastest data flow. Furthermore, the system naturally remains resilient against local congestions. It is also always operational without detecting failures and switching from failed replicas to functioning replicas.

A. Contributions

The contributions of this paper are as follows:

1) We propose a new replication framework for fast and robust stream processing in wide area networks.
2) We define replication transparency as the key concept for our replication framework. We also devise stream-processing primitives for replication transparency.
3) We develop an algorithm for managing replicas. This algorithm strives to achieve the best latency guarantee, relative to the cost paid for replication.
4) We demonstrate the utility of our work through experiments on PlanetLab [13] and trace-driven simulations.

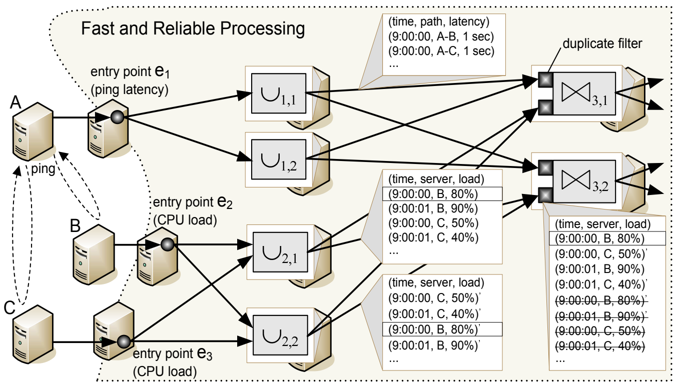


Fig. 1. An Example of Replication – Server A continuously measures the latencies of connections to servers B and C and reports them via entry point e_1. Servers B and C report their CPU usage via entry points e_2 and e_3, respectively. Replicas $\cup_{2,1}$ and $\cup_{2,2}$ merge streams from e_2 and e_3 and feed $\bowtie_{3,2}$ in parallel. $\bowtie_{3,2}$ uses whichever tuple arrives first from $\cup_{2,1}$ and $\cup_{2,2}$, while ignoring duplicates (i.e., late tuples; see those stroked-through) .

B. Road map

The rest of this paper is organized as follows. We give an overview in Section II and design a semantic model for replicated stream processing in Section III. In Section IV, we devise stream-processing primitives according to our replication model. Next, we discuss replica management in Section V and show experimental results in Section VI. We present related work in Section VII and conclude in Section VIII.

II. Background

In this section, we describe the assumptions behind our work and introduce our replication framework. Then, we stress the specific problems tackled in this paper.

A. Assumptions

System. We assume a wide area network as the substrate for stream processing. We assume that the network has abundant computation and communication resources.

Communication. We assume that the network layer runs a reliable, in-order, point-to-point message delivery protocol such as TCP.

Failure Model. We assume fail-stop server/network failures. We do not consider Byzantine failures where faulty components can behave in arbitrarily erroneous ways.

Query. We assume that queries are translated into a directed acyclic graph of operators [14], [2], [3]. In this paper, we consider the stream-processing operators defined in [14].

B. The Basic Architecture

To manage the system in a scalable fashion, we group servers into logical clusters each of which comprises tens of servers. For each cluster to autonomously handle queries that span distant stream sources and applications, each cluster includes servers at *diverse* locations rather than those only within a small area. For this reason, clusters may overlap significantly with each other in terms of their geographic coverage. Hereafter, we focus on replicating operators within a logical cluster.

As illustrated in Fig. 1, our approach guarantees fast and reliable processing by making multiple replicas feed each downstream replica (observe that replicas $\cup_{2,1}$ and $\cup_{2,2}$ both send data to the second input of $\bowtie_{3,2}$). To achieve reliable processing with unstable stream sources, we introduce entry points, which serve as the starting points for reliable stream processing. Entry points buffer input tuples from external stream sources until they safely arrive at the downstream replicas. They also replicate input tuples at other peer entry points to improve availability. Finally, entry points have well-synchronized clocks [15] and can timestamp input tuples on behalf of unsynchronized stream sources.

C. Problem Statements

To complete the replication framework described above, we need to address the following research challenges.

1) What *execution semantics* should we consider for replicated stream processing?
   We take the position that replication must be *transparent* to users. In other words, we want to guarantee that replicated processing (with failures and delays) always produces the results that would appear without replication and failures. We discuss the details in Section III.
2) How should we *extend processing primitives* for replication transparency?
   A simple way to ensure replication transparency would be to identically execute peer replicas. As shown in Section VI, however, such an approach introduces extra delays. Therefore, we let replicas run differently as long as replication transparency is achievable. To produce correct results, we also need to remove duplicates, as shown in Fig. 1. In Section IV, we design processing primitives in this fashion.
3) How should we manage replicas?
   As demonstrated in Section VI, the deployment of replicas significantly affects the network cost, performance (in terms of the average end-to-end latency), and availability (in terms of the probability that the system delivers results to end-users within a certain latency). In Section V, we discuss managing replicas so as to achieve the best latency given a fixed replication cost.

III. Replication Transparency

The central notion behind our work is *replication transparency*. Under this guarantee, each application always receives tuples that would be generated in the *ideal non-replication scenario* where the system is completely free from failures and delays. Because timestamps of tuples are usually used as essential elements for processing [14], replication transparency also requires the timestamps to represent the precise times when the tuples would be generated in the ideal non-replication scenario. Our goal in this paper is to deliver such results to applications as soon as possible, by use of replication.

The definition of replication transparency above is specified only in terms of *the results to applications*. A natural question at this point is how we should instrument operator replicas to produce such results in the end. We first take the position that each operator replica must generate the tuples that would appear in the ideal non-replication scenario. We, however, let each replica run independently, while processing any available data. As demonstrated in Section VI, this relaxation expedites processing, compared to previous approaches that execute replicas identically at distant servers. This, however, causes multi-way operators, such as Union and Join, to generate tuples in a random order. In Fig. 1, replicas $\cup_{2,1}$ and $\cup_{2,2}$ produce outputs in different orders for this reason.

Despite the complications above, we achieve replication transparency as follows:

1) *We merge stream replicas into a non-duplicate stream using a non-blocking filter.* In our replication framework, each operator replica receives inputs from multiple upstream replicas. An operator such as a count aggregate [14], however, may produce incorrect results if it processes duplicate tuples. Furthermore, processing duplicates would waste CPU cycles. In Section IV-B, we devise non-blocking filters that eliminate duplicates from disordered stream replicas. In Fig. 1, the second duplicate filter of $\bowtie_{3,2}$ merges two stream replicas into a non-duplicate stream that again contains the same tuples as its input streams. Because the input streams of the filter have different orders, and because the filter operates in a non-blocking fashion, the output of the filter has a new different order.
2) *We sort disordered streams only when necessary.* Order-sensitive applications and operators, such as those with count-based windows [14], must process inputs in the order of the ideal non-replication scenario. In Section IV-C, we discuss sorting streams, while introducing extra delays. Due to this penalty, we sort streams only when necessary. In other words, we bypass the sorting phase for order-insensitive operators and applications.
3) *We redesign operators so that they can produce, from disordered input streams, the output tuples that would appear in the ideal non-replication scenario.* The output stream need not be ordered, because downstream replicas can handle disorder. We call this property of operators *replica consistency* because it guarantees that replicas always produce consistent output streams from consistent input streams. We say that two streams are *consistent* if they contain the same tuples regardless of internal order. In Sections IV-D through IV-F, we devise non-blocking implementations of Filter, Map, Union, and Join and a blocking implementation of Aggregate. All these operators guarantee replica consistency.

The arguments above state that we can achieve replication transparency by eliminating duplicates, minimally sorting data streams, and making every operator ensure replica consistency. Replica consistency can be defined formally as follows:

Definition 1: Infinite streams S and S' are consistent (denoted by $S \equiv S'$) if they contain the same tuples regardless of internal order. Specifically, $S \equiv S'$ if there exists a permutation $\mu : \mathbb{N} \rightarrow \mathbb{N}$ such that $S[i] = S'[\mu(i)]$, where $S[i]$ denotes the ith tuple in stream S.

Definition 2: **(Replica Consistency)** Let $o(S_1, S_2, \cdots, S_n)$ denote the set of all possible output streams that operator o with n inputs can generate from input streams $S_1, S_2, \cdots, S_n$. Then, we say that operator o guarantees replica consistency if any possible output streams $O \in o(S_1, S_2, \cdots, S_n)$ and $O' \in o(S'_1, S'_2, \cdots, S'_n)$ are consistent (i.e., $O \equiv O'$) for any consistent input streams S_i and S'_i (i.e., $S_i \equiv S'_i, 1 \leq i \leq n$).

IV. Extension for Replication Transparency

In this section, we devise the processing primitives for replication transparency. In Section IV-A, we introduce punctuations because many of our primitives use them. Next, we discuss filtering out duplicates in Section IV-B and sorting streams in Section IV-C. In Sections IV-D through IV-F, we extend stream-processing operators for replica consistency. Any operator, including user-defined ones, can be used in our replication framework if it guarantees replica consistency.

A. Management of Punctuations

In our approach, either stream sources or their downstream entry points timestamp tuples using well-synchronized clocks. They also periodically send special values, called punctuations [16], [12]. Punctation p in input stream S guarantees that the timestamp of any subsequent tuple in S will be larger than p. All streams in the system can also satisfy this property if each operator forwards p as soon as it receives p via all its inputs. This is because an operator in that situation will always receive tuples with timestamps larger than p. This implies that, in the ideal non-replication scenario, the operator would process all these tuples after time p, and thus the timestamps of all later output tuples must be larger than p.

B. Duplicate Filtering

As pointed out in Section III, we use duplicate filters to eliminate duplicate tuples from stream replicas. Duplicate filters must deal with disorder and multiple occurrences of the same tuple in each stream replica. They also should not block the data flow. Algorithm 1 describes the operation of our duplicate filter. For tuple t from stream replica S_i, we first check if t satisfies the condition in line 2. Otherwise (i.e., if $t.\text{timestamp} \leq \text{max\_punctuation}$), t is a duplicate because the filter received the current maximum punctuation (max_punctuation) from a stream replica before. This implies that the filter already received, from the same stream replica, all tuples t' such that $t'.\text{timestamp} \leq \text{max\_punctuation}$.

Next, the filter uses a variable $\text{count}[t][i]$ to remember how many times it received tuple t from stream replica S_i (line 3). If the filter received t from S_i more times than any other stream replica, it passes t to the operator as a non-duplicate (line 5). Otherwise (i.e., if $\exists S_j$ such that $\text{count}[t][i] \leq \text{count}[t][j]$), t is a duplicate because the filter already received a corresponding tuple from S_j. Lines 7-11 describe that, whenever a new punctation arrives, the filter

Algorithm 1: Duplicate Filtering

```
1  whenever tuple $t$ arrives from stream replica $S_i \in \{S_j\}_{j=1}^{k}$ do
2      if $t$.timestamp $>$ max_punctuation then
3          count[$t$][$i$] $\leftarrow$ count[$t$][$i$] + 1;
4          if count[$t$][$i$] $>$ max_count[$t$] then
5              output($t$);
6              max_count[$t$] $\leftarrow$ count[$t$][$i$];
7  whenever punctuation $p$ arrives from any stream replica do
8      if $p >$ max_punctuation then
9          output($p$);
10         max_punctuation $\leftarrow p$;
11         remove all count[$t$][$*$] such that $t$.timestamp $\leq p$;
```

can safely remove count variables for all the known duplicate tuples. The life time of each count variable is thus bounded by the punctuation interval.

The following theorem proves the correctness of Algorithm 1.

Theorem 1: Let $\mathfrak{D}(S_i)_{i=1}^{k}$ denote the set of all output streams that duplicate filter $\mathfrak{D}$ can produce from stream replicas $\{S_i\}_{i=1}^{k}$. If $\{S_i\}_{i=1}^{k}$ are consistent, any $O \in \mathfrak{D}(S_i)_{i=1}^{k}$ is also consistent with them (i.e., $O \equiv S_i$, $1 \leq i \leq k$).
Proof: Since $\{S_i\}_{i=1}^{k}$ are consistent, all of them commonly contain an arbitrary tuple t the same number of times (say m). It suffices to prove that $\mathfrak{D}$ passes t, m times in any case. (1) if $m = 0$, $\mathfrak{D}$ cannot pass t since none of the streams $\{S_i\}_{i=1}^{k}$ contains t. (2) If $m > 0$, without loss of generality, suppose that $\mathfrak{D}$ receives the mth t from S_1 before it receives the mth t from S_j $(2 \leq j \leq k)$. Since this implies that $\mathfrak{D}$ has not yet received punctuation $p \geq t.\text{timestamp}$ from any stream replica, t must satisfy the condition in line 2. Since it also implies that $\text{count}[t][1] > \text{count}[t][j]$ $(2 \leq j \leq k)$, t must satisfy the condition in line 4. Therefore, D must pass t to the operator. This also must be the mth output of t because, by the induction hypothesis, $\mathfrak{D}$ must have passed t, $(m-1)$ times before this. $\mathfrak{D}$ then sets both $\text{count}[t][1]$ and $\text{max\_count}[t]$ to m. Since $\text{count}[t][j] \leq \text{max\_count}[t] = m(2 \leq j \leq k)$, $\mathfrak{D}$ will filter out t afterwards . $\square$

C. Sorting Streams

If a stream S feeds an order-sensitive operator/application, we sort S using punctuations. We first insert each tuple from S into a sorted list $\mathfrak{L}$. This list contains tuples in the order of increasing timestamps. If multiple tuples have the same timestamp, we enforce a unique ordering, for example, by regarding these tuples as byte arrays and radix-sorting them. Whenever a punctuation p arrives from stream S, we remove tuples $\{t \in \mathfrak{L} : t.\text{timestamp} \leq p\}$ from $\mathfrak{L}$, while passing them, in the sorted order, to the next operator/application. This sorting phase each time observes the entirety of the tuples in S up to a punctuation p and passes them in a unique order. Therefore, it allows all peer replicas to process inputs in the same order, with extra delays that depend on the punctuation interval. This sorting phase is described in Algorithm 4 as part of an order-sensitive operator (refer to lines 1-6).

Algorithm 2: Join

```
1  whenever tuple $t_1$ arrives at input $I_1$ do
2      for each $t_2 \in B_2$ such that
       $|t_1.\text{timestamp} - t_2.\text{timestamp}| < w \wedge P(t_1, t_2)$ do
3          output($t_1 \otimes t_2$);
4      $B_1$.add($t_1$);
5  whenever punctuation $p_1$ arrives at input $I_1$ do
6      $B_2$.remove ($\{t_2 \in B_2 : p_1 - t_2.\text{timestamp} > w\}$);
   * Inputs to $I_2$ are processed symmetrically
```

In the rest of this section, we discuss extending operators for replica consistency.

D. Stateless Operators and Replica Consistency

Stateless operators are those that produce each output tuple based on only the last input tuple [14]. Filter forwards each input tuple if the tuple satisfies a pre-defined predicate. Map converts each input tuple into a different tuple. Union merges two or more streams into a single output stream.

Each stateless operator naturally guarantees *replica consistency* because it processes each tuple deterministically, regardless of the input order. In detail, all replicas of a Filter must pass a tuple if it satisfies the predicate. All replicas of a Map must identically convert each input tuple and replicas of a Union must forward each input tuple. Therefore, our replication framework uses stateless operators as defined in [14]. Filter, Map, Union are all *non-blocking*. Union is *nondeterministic* because, as demonstrated by $\cup_{2,1}$ and $\cup_{2,2}$ in Fig. 1, its output order can vary depending on the arrival order of tuples across its input streams.

E. Extending Join for Replica Consistency

Join [14] has two inputs I_1 and I_2, window size w, and predicate P. For any input tuples t_1 from I_1 and t_2 from I_2, it outputs the concatenation $t_1 \otimes t_2$ of them if they (a) belong to the same time window (i.e., $|t_1.\text{timestamp} - t_2.\text{timestamp}| < w$) and (b) satisfy predicate P (i.e., $P(t_1, t_2)$ holds). Here, we do not consider the extra features, called slack and timeout, of Join that handle disorder and silence. This is because our approach tackles these issues by use of punctuations, while expediting processing through replication.

We implement Join as illustrated in Algorithm 2. Lines 1-4 output the concatenation of matching input tuples, while using B_i to buffer the tuples entered input I_i. The timestamp of $t_1 \otimes t_2$ is set to $\max(t_1.\text{timestamp}, t_2.\text{timestamp})$, which is the time when the tuple would be produced in the ideal non-replication scenario. Lines 5-6 discard the buffered tuples that will no longer be used. Any tuple $t_2 \in B_2$ that satisfies the condition in line 6 cannot match with any tuple t_1 that will arrive at I_1. This is because $t_1.\text{timestamp} - t_2.\text{timestamp} > p_1 - t_2.\text{timestamp} > w$.

This Join implementation is *non-blocking* because it produces each output tuple as soon as it obtains both constituent input tuples. It guarantees *replica consistency* because, for

Algorithm 3: Aggregate with Time Windows

```
1 whenever tuple $t$ arrives do
2     for each window $w \in \mathcal{G}(t).\mathcal{W}(t.\text{timestamp})$ do
3         $w.\text{update}(t)$;

4 whenever punctuation $p$ arrives do
5     for each $g \in \mathcal{G}$ do
6         for each $w \in g.\text{windows}$ such that $w.\text{expir\_time} \leq p$ do
7             $g.\text{windows.remove}(w)$;
8             $\text{output}(w.\text{get\_summary}())$;
```

Algorithm 4: Aggregate with Count-based Windows

```
1 whenever tuple $t$ arrives do
2     $\mathfrak{L}.\text{insert}(t)$; // sorted list

3 whenever punctuation $p$ arrives do
4     while $L \neq \emptyset \wedge \mathfrak{L}.\text{first}().\text{timestamp} \leq p$ do
5         $\text{agg\_count\_windows}(\mathfrak{L}.\text{first}())$;
6         $\mathfrak{L}.\text{remove\_first}()$;

7 $\text{agg\_count\_windows}(t)$
8 begin
9     count++;
10    for each window $w \in \mathcal{G}(t).\mathcal{W}(\text{count})$ do
11        $w.\text{update}(t)$;
12        if $w.\text{expir\_count} \leq \text{count}$ then
13            $g.\text{windows.remove}(w)$;
14            $\text{output}(w.\text{get\_summary}())$;
15 end
```

each pair of matching input tuples, it produces the concatenation of them exactly once, regardless of the inter-arrival order of the input streams. Similar to Union, Join is *nondeterministic* and may introduce *disorder*.

F. Extending Aggregate for Replica Consistency

Aggregate [14] splits input stream I into substreams $\{I[g]\}_{g \in \mathcal{G}}$, where $\mathcal{G}$ is the set of groups and $I[g]$ is a subsequence of I that contains tuples belonging to group g. For each substream $I[g]$, this operator forms windows (sets of tuples) based on either *timestamps* or *the count of tuples*. If a window expires, Aggregate produces an output tuple computed from the tuples in the window. For the reasons described in Section IV-E, we do not consider slack and timeout.

For *replica consistency*, Aggregate must form and close windows uniquely, despite disorder in the input stream.

Aggregate with Time Windows. In this case, each group g forms windows of w seconds every s seconds. Therefore, for input tuple t, we 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), we get $\mathcal{W}(\text{9:00:43}) = \{[\text{9:00:35, 9:00:45}), [\text{9:00:40, 9:00:50})\}$. Lines 1-3 in Algorithm 3 uniquely form windows regardless of the input order. Then, lines 4-8 use punctuations to find the windows that cannot contain more tuples. This allows Aggregate to produce the same output tuples from any consistent input stream.

Aggregate with Count-based Windows. This operator uses, for each group g, a window of w tuples that skips s tuples whenever it moves. Because this operation is order-sensitive, we sort input stream as described in Section IV-C (lines 1-6 in Algorithm 4). After this, the operator forms and closes windows (lines 7-15) using the count of input tuples.

The two implementations above are *blocking* because they wait for punctuations to assure that they obtained all the required tuples. With time windows, we can minimize the delay by making stream sources or their entry points produce punctuations at expiration times of windows. With count-based windows, the delay depends on the punctuation interval.

V. Management of Replicas

As demonstrated in Section VI, the deployment of replicas affects latency guarantees as well as resource usage. In this section, we discuss managing replicas to *achieve the best latency guarantee, relative to a fixed replication cost*. In Section V-A, we discuss deploying replicas initially in a resource-efficient fashion. In Section V-B, we devise an algorithm that reduces the resource usage to a target level, while minimally degrading the latency guarantee. For this, the algorithm finds the least useful stream/operator replicas each time and discards them. In Section V-C, we consider reviving garbage-collected replicas to cope with changes in system conditions.

A. Deployment of Replicas

As illustrated in Section II-B, our replication framework forms logical clusters each of which comprises servers at diverse locations. Servers in the same cluster elect a coordinator for them. In this subsection, we discuss how the coordinator of cluster $\mathfrak{S}$ should deploy a predefined number ($k_{\max}$) of replicas for each operator. Our strategy strives to *minimize the overall network cost* similarly to operator placement approaches in the non-replication context [17], [18]. In our replication framework, for a collection of stream replicas $\mathfrak{R}$, the network cost of $\mathfrak{R}$, $\text{cost}(\mathfrak{R})$, is defined as the sum of individual stream replicas' network costs. Formally, $\text{cost}(\mathfrak{R}) = \sum_{S \in \mathfrak{R}} \text{cost}(S)$ where $\text{cost}(S)$ denotes the cost of stream replica S. $\text{cost}(S)$ is in turn defined as $\text{rate}(S) \cdot \text{latency}(S)$ where $\text{rate}(S)$ and $\text{latency}(S)$ are the data rate and the network latency of stream replica S, respectively. This *bandwidth-delay product* is based on the idea that the longer data stays in the network, the more resources it tends to use. An optimal deployment under this metric also tends to choose fast network links, thereby accomplishing low-latency processing.

The Replica Deployment Algorithm. Algorithm 5 describes our replica deployment strategy. Each of the $k_{\max}$ deployment phases creates, for each operator o in query $\mathfrak{Q}$, a new replica on a server that minimally increases the network cost. For a new replica, it first finds good candidate servers $\mathfrak{C}$ (lines 6-7). Such a server $\mathfrak{s}$ must not be busy (line 6). $\text{load}(\mathfrak{s})$ and $\text{load}(o)$ are the current load of server $\mathfrak{s}$ and the expected load of the new replica of o, respectively. $\text{capacity}(\mathfrak{s})$ is the processing capacity of server $\mathfrak{s}$. With $\alpha < 1$, the condition in line 6 checks if $\mathfrak{s}$ is likely to have enough available CPU cycles even if it runs the new replica. A good candidate server $\mathfrak{s}$ also must have a low risk of falling into the same network partition with

Algorithm 5: Replica Deployment (for query $\mathfrak{Q}$)

```
1  for $i = 1$ to $k_{\max}$ do
2      deploy($\mathfrak{Q}$);

3  deploy($\mathfrak{Q}$)
4  begin
5      for each operator $o \in \mathfrak{Q}$ do
6          $\mathfrak{C} \leftarrow \{\mathfrak{s} \in \mathfrak{S} : \text{load}(\mathfrak{s}) + \text{load}(o) < \alpha \cdot \text{capacity}(\mathfrak{s}) \wedge$
7                  $\min(\text{latency}(\mathfrak{s}, \mathfrak{s}')) > d_{\min}, \forall \mathfrak{s}' \in \mathfrak{S}(o)\}$;
8          Find $\mathfrak{s}^* \in \mathfrak{C}$ such that $\forall \mathfrak{s} \in \mathfrak{C}$
9                  $\text{cost}_{\mathfrak{s}^*}(\text{in}(o) \cup \text{out}(o)) \leq \text{cost}_{\mathfrak{s}}(\text{in}(o) \cup \text{out}(o))$;
10         $\mathfrak{s}^*$.deploy($o$);
11 end
```

any $\mathfrak{s}'$ of servers $\mathfrak{S}(o)$ that currently run replicas of o. To ensure this, the heuristic in line 7 uses network latencies to ensure that replicas are always deployed on sufficiently distant servers. In several test cases, it turned out that 30ms to 70ms is a good range for the minimum latency $d_{\min}$ between peer replicas. After finding candidate servers $\mathfrak{C}$, the coordinator chooses the server $\mathfrak{s}^*$ that will minimize the network cost of the input streams $\text{in}(o)$ and output streams $\text{out}(o)$ of the new replica (lines 8-9). Specifically, $\text{in}(o)$ denotes all possible input streams from the replicas directly upstream from any replica of o, and $\text{out}(o)$ denotes all possible output streams to the replicas directly downstream from any replica of o. In line 9, $\text{cost}_{\mathfrak{s}}(\text{in}(o) \cup \text{out}(o))$ represents the network cost of these input and output stream replicas, provided that server $\mathfrak{s}$ runs the new replica. Formally, $\text{cost}_{\mathfrak{s}}(\text{in}(o) \cup \text{out}(o)) = \sum_{S \in \text{in}(o)} \text{rate}(S) \cdot \text{latency}(S.\text{source}, \mathfrak{s}) + \sum_{S \in \text{out}(o)} \text{rate}(S) \cdot \text{latency}(\mathfrak{s}, S.\text{destination})$. Finally, the coordinator deploys the new replica on $\mathfrak{s}^*$ (line 10).

Discussion. Our strategy above strives to find, phase-by-phase, the deployment that will minimally increase the network cost while providing the desired availability level. It is, however, hard to find the optimal deployment in the first phase because the data rate of each stream and the processing load of each operator are not yet known (these statistics are available in the later phases). The network cost also changes over time as the data rates and latencies of streams vary. For this reason, we use an approach that initially deploys replicas aggressively and dynamically garbage-collects/revives them afterwards.

B. Garbage Collection

Our replica deployment algorithm creates $k_{\max}$ replicas for each operator. Between $k_{\max}$ upstream replicas and $k_{\max}$ downstream replicas, it also creates $k_{\max}^2$ stream replicas. Although $k_{\max}$ is usually set to a small number (say 4 or 5 at most) in practice, using all these stream replicas would waste system resources. Furthermore, faster operator replicas and those that feed many downstream replicas would have a higher impact on processing than others. Thus, we use a strategy that periodically discards the least useful stream and operator replicas. This is to reduce the resource usage, while minimally degrading the latency guarantee. To maintain the minimum fault-tolerance level, however, we ensure that at least $k_{\min}$ replicas of each operator survive.

Algorithm 6: Garbage Collection (for stream replicas $\mathfrak{R}$)

```
1  whenever $\sum_{S \in \mathfrak{R}} \text{cost}(S) > \theta$ do
2      $\mathfrak{C} \leftarrow \{\text{dependents}(S) : S \in \mathfrak{R}\} - \{\emptyset\}$;
3      Find a collection of stream replicas $\mathfrak{D}^* \in \mathfrak{C}$ such that
           $\frac{\text{utility}(\mathfrak{D}^*)}{\text{cost}(\mathfrak{D}^*)} \leq \frac{\text{utility}(\mathfrak{D})}{\text{cost}((\mathfrak{D})}, \forall \mathfrak{D} \in \mathfrak{C}$;
4      discard($\mathfrak{D}^*$);

5  dependents($S$)
6  begin
7      return dependents($S, \emptyset$);
8  end

9  dependents($S, \mathfrak{D}$)
10 begin
11     $\mathfrak{D}$.add($S$);
12     $\mathfrak{D} \leftarrow$ dependents($S$.source, $\mathfrak{D}$);
13     if $\mathfrak{D} = \emptyset$ then
14         return $\emptyset$;
15     else
16         return dependents($S$.destination, $\mathfrak{D}$);
17 end

18 dependents($\mathfrak{o}, \mathfrak{D}$)
19 begin
20     if $|\mathfrak{C}(\mathfrak{o}; \mathfrak{D})| < k_{\min}$ then
21         return $\emptyset$;
22     if need_to_remove($\mathfrak{o}, \mathfrak{D}$) then
23         for each $S \in \text{in}(\mathfrak{o}) \cup \text{out}(\mathfrak{o}) - \mathfrak{D}$ do
24             $\mathfrak{D} \leftarrow$ dependents($S, \mathfrak{D}$);
25             if $\mathfrak{D} = \emptyset$ then
26                 return $\emptyset$;
27     return $\mathfrak{D}$;
28 end
```

The Garbage-Collection Algorithm. Algorithm 6 illustrates our garbage-collection strategy. Periodically (every 5 minutes in our prototype), the coordinator computes the current overall network cost. If the cost is higher than a target utilization level θ (line 1), it finds the group of least useful replicas, relative to the network cost paid for them (lines 2-3). It then asks the related servers to discard them (line 4). $\text{dependents}(S)$ in line 2 finds the group of replicas that must be discarded together with stream replica S. For example, if an operator replica $\mathfrak{o}$ has only one output stream S, removing S will make $\mathfrak{o}$ useless and therefore necessitates removing all the input streams of $\mathfrak{o}$ as well. In this case, $\text{dependents}(S)$ must contain the input streams of $\mathfrak{o}$. Given replicas $\mathfrak{D}$ to discard, lines 09-17 check if removing stream replica S will require removing its source replica (line 12) or destination replica (line 16) for the reason above. Lines 13-14 handle the case where we cannot remove S because we would have fewer than $k_{\min}$ operator replicas if S was removed (see also lines 20-21). Lines 18-28 are to check an operator replica $\mathfrak{o}$. Lines 20-21 are to keep at least $k_{\min}$ operator replicas (in line 20, $\mathfrak{C}(\mathfrak{o}; \mathfrak{D})$ represents the number of replicas of o if we remove replicas in $\mathfrak{D}$). Line 22 checks if removing replicas in $\mathfrak{D}$ will make operator replica $\mathfrak{o}$ useless and thus require removing $\mathfrak{o}$. If so, the algorithm visits the input and output streams of $\mathfrak{o}$ while recursively applying the algorithm (lines 23-26). If removing such streams leads to having fewer than $k_{\min}$ replicas for some operator (line 25), it decides not to remove any streams (line 26).

Measuring the Utility of Each Replica. As shown above, our garbage-collection algorithm considers the utility of each stream. Intuitively, we define utility as the impact of a replica on the data flow towards applications. To compute this, our heuristic first uses duplicate filters to measure the contributions of stream replicas to the next operator. Specifically, each duplicate filter gives, for each input tuple, weights $\frac{1}{1}, \frac{1}{2}, \frac{1}{3}, \cdots$ to its input stream replicas based on how early they deliver the tuple (the fastest replica gets the highest weight each time). These weights, however, do not capture the impact after the next operator. Thus, our heuristic periodically (every 30 seconds in our prototype) computes the utilities of stream/operator replicas from applications to more upstream replicas. Specifically, for a group of stream replicas $\{S_i\}_{i=1}^{k}$, $\text{utility}(S_i)$, the utility of S_i, is computed as $\frac{w(S_i)}{\sum_{j=1}^{k} w(S_j)}\text{utility}(\mathfrak{o})$ where $w(S)$ is the accumulated weight of stream S and $\text{utility}(\mathfrak{o})$ is the utility of the operator replica $\mathfrak{o}$ that $\{S_i\}_{i=1}^{k}$ commonly feed. $\text{utility}(\mathfrak{o})$ is set to 1 if $\mathfrak{o}$ is an application. Otherwise, it is computed as $\sum_{S \in \text{out}(\mathfrak{o})} \text{utility}(S)$ where $\text{out}(\mathfrak{o})$ denotes the output streams of $\mathfrak{o}$.

Discussion. As described above, our garbage-collection algorithm victimizes replicas with high cost and small contribution to downstream processing. Replicas of the opposite kind are likely to survive over time. Such surviving replicas are in general those with high popularity (i.e., those eventually connected to a large number of applications), those upstream, and those along fast data flows.

C. Adaptation to Changes

Our garbage-collection algorithm saves resources while striving to preserve the latency guarantee. If failures or local congestions occur, however, the surviving replicas may experience unexpected delays. We solve this problem by reviving garbage-collected replicas. In detail, if an operator replica observes delays longer than a threshold (say 10 seconds) across its input streams, it first finds garbage-collected input streams that connect to functioning upstream replicas. If there is such one, it revives the stream replica. Otherwise, it revives a garbage-collected upstream operator replica and acquires a new connection from that one. If the problem persists, we can either create more upstream replicas, revive/create peer replicas that could replace the hindered replica, or simply discard the hindered one.

If the coordinator finds that the current network cost is lower than the target level θ, it can also assist hindered replicas as described above.

VI. Experimental Results

In this section, we present various results that substantiate the utility of our work. In Section VI-A, we describe how we set up the experiments and simulations. Then, we present results obtained from our prototype (Sections VI-B through VI-D) and a trace-driven simulator (Sections VI-E and VI-F).

A. The Setup

In all of our experiments, we assumed a system configuration where servers are grouped into clusters each of which consists of 30 servers. Thus, for our prototype, we chose 30 distant PlanetLab servers [13] that reliably communicate with others. The results obtained, however, varied each time because the servers were used intensively by many users. To compare various approaches under an identical condition, we also conducted simulations. Our prototype and simulator use the same source code except for the communication component. To send tuples and invoke remote procedures, our prototype uses TCP sockets. The simulator instead emulates network delays using a trace. We obtained this trace by recording actual network delays between 100 PlanetLab servers every 10 seconds for a month starting from February 13, 2007.

Our experiments, except that in Section VI-D, used the query illustrated in Fig. 1, while adding Filters and applications after the Joins. To easily detect data loss, however, we used stream sources that periodically generated tuples. We also made the Filters always pass their inputs. In detail, each server ran two kinds of stream sources, one that reported the server's CPU load and the other that reported the latencies of connections to other servers. Such input streams were first merged at Unions, one for each input type, and the subsequent Joins correlated load and latency readings. For the experiments in Sections VI-B and VI-D, stream sources generated input tuples every half a second and 1 millisecond, respectively. In the other experiments, input tuples were generated every 10 seconds. Joins in Sections VI-B and VI-D used time windows of half a second and 100 milliseconds, respectively. In other cases, the window size was set to 10 seconds.

Given the query above, we deployed replicas. In Sections VI-B and VI-D, we manually did the task. In other cases, the coordinator first obtained statistics on network delays between servers and the data rates of streams through a test run. Then, it deployed operators in a non-replicated fashion, using the spring relaxation algorithm [18]. This was to start with the best operator placement that has the lowest network cost. After this, the coordinator replicated operators and streams, according to the chosen replication method.

B. Comparison of Techniques for Reliable Stream Processing

In this experiment, we compare our replication technique with previous high-availability techniques. For this, we manually placed $\cup_{1,1}$ at WISC, $\cup_{2,1}$ at Purdue, and $\bowtie_{3,1}$ at OSU.

In Fig. 2, the curve labeled "no replication" represents how the latency, without replication, varied over time at the output of $\bowtie_{3,1}$. The latency of a tuple was defined as the difference between the wall-clock time and the timestamp of the tuple (i.e., the time when the tuple would be produced in the ideal non-replication scenario, equivalently the earliest time when the tuple could be generated). In this experiment, we crashed the stream-processing engine at WISC at time 60. After that, $\bowtie_{3,1}$ did not produce any output because it no longer received tuples from $\cup_{1,1}$. In contrast, other curves show that reliability techniques indeed provide protection against failures.

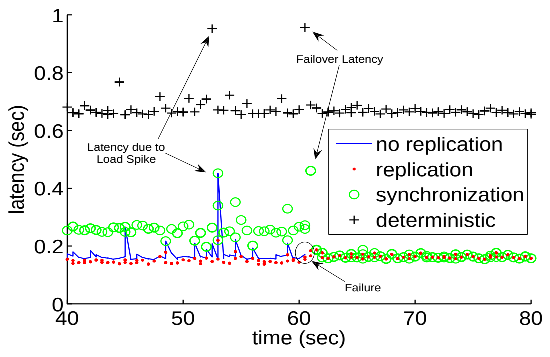


Fig. 2. Comparison of Reliability Techniques

The curve labeled "replication" shows how our replication technique behaved when we added replicas $\cup_{1,2}$ and $\cup_{2,2}$ at Purdue and WISC, respectively. In this case, despite the failure at WISC, the processing continued relying on $\cup_{1,2}$ and $\cup_{2,1}$ at Purdue. After the failure, however, the latency increased because $\bowtie_{3,1}$ no longer benefited from replication. In Fig. 2, "synchronization" shows the performance of a previous technique that always enforces an identical execution between primaries and backups [9]. In this technique, each primary sends extra information, called *determinants*, to the backup so that the backup can mimic the processing of the primary. For Unions, we used the inter-arrival order of input tuples to generate determinants. This method introduces extra delays because primaries must hold output tuples until backups acknowledge the recept of determinants. Checkpoint-based techniques [9], [10] also show similar behavior because primaries hold outputs until one round of checkpoint finishes. After the failure, the latency dropped since $\cup_{1,2}$ and $\cup_{2,1}$ at Purdue no more had synchronization partners.

Finally, "deterministic" shows the variation of latency under a method that runs peer replicas (e.g., $\cup_{1,1}$ and $\cup_{1,2}$) identically by feeding the replicas in the same order [12]. As described in Section IV-C, sorting streams introduces extra delays because tuples are held until a relevant punctuation arrives. We can reduce extra delays by more frequently producing punctuations. The pace of $\bowtie_{3,1}$, however, is eventually determined by the slowest input flow.

In summary, the figure shows that our replication technique improves both *performance* and *reliability* because it always benefits from the best of multiple independent data flows. On the other hand, previous approaches degrade performance because they identically run replicas at distant servers, thereby introducing extra delays. In previous approaches, the failure of a server also disrupts the processing until the downstream servers notice it and switch to another upstream server (observe the failover latencies in Fig. 2).

C. Impact of Replication on Latency

Fig. 3 shows how the end-to-end latency at an application varies over time depending on the degree of replication. In this experiment, each server ran three stream-processing engines and used each of them for a different degree of replication. For example, the curve labeled "$k_{\max}$=3" shows the latency

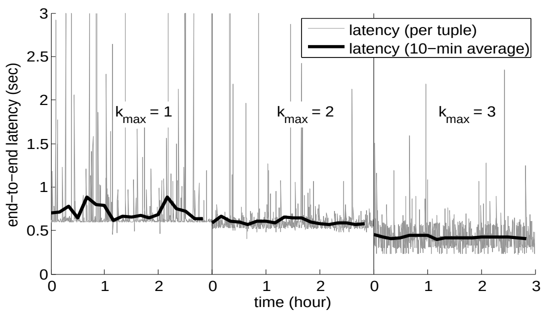


Fig. 3. Impact of Replication on Latency

operators	no replication	degree of replication			
		1	2	3	4
Union	0.7	1.1 (1.43x)	1.8 (2.57x)	2.7 (3.86x)	3.5 (5.00x)
Filter	0.8	1.2 (1.38x)	1.9 (2.37x)	2.8 (3.50x)	3.6 (4.50x)
Aggregate	2.3	2.7 (1.17x)	3.6 (1.57x)	4.4 (1.92x)	5.2 (2.26x)
Join	9.5	10.3 (1.08x)	10.8 (1.14x)	11.6 (1.22x)	12.3 (1.29x)

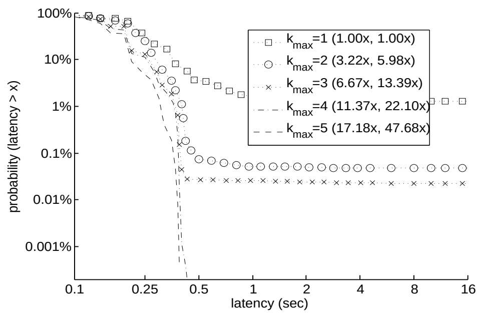
TABLE I
CPU COST OF A REPLICA (% CPU CYCLES)

results obtained from the engines that collectively deployed 3 replicas for each operator as described in Algorithm 5. The figure shows that the average as well as the variance of latency decrease as we deploy more replicas. This is because each operator in the system is provided more input flows and thus can benefit from better ones.

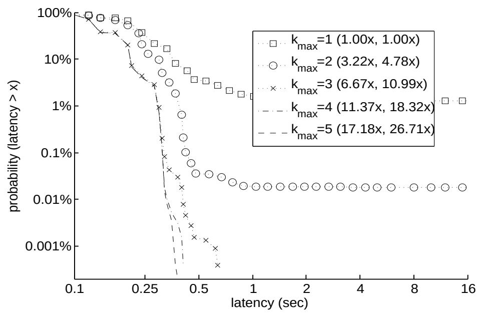
D. CPU cost for Replications

Using our prototype, we also measured the CPU cost for replication, using AMD Sempron 2800+ CPUs. In this experiment, we fed 1K tuples/sec to each input of the operators. The operators were instrumented to output 1K tuples/sec as well. Specifically, the Filter always passed input tuples after evaluating the predicate and the Aggregate computed the count of input tuples using a window of 10ms that slid every 1 ms. The Join matched each input tuple with 100 input tuples on the other input, but produced only one output tuple every 1 ms as the result of predicate evaluation.

For each operator type, we first fixed the degree of replication ($k_{\max}$) to 4 and gradually added replicas until approximately half the cycles of a CPU were used. After this, we decreased the degree of replication from 4 to 1, while finding the per-replica CPU cost by dividing the CPU usage by the number of replicas. We also measured the CPU cost for the non-replication case. Table I summaries the results. The "no replication" column shows that each operator has a different processing cost. Each row of the table shows that the CPU cost per replica increases as we add more input/output stream replicas. This increase in the CPU cost corresponds to the overheads of removing duplicates as well as sending and receiving tuples via stream replicas. Finally, the table shows that the per-replica CPU cost increases at a different pace for each operator. The Join operator has the lowest growth rate



(a) random



(b) min-cost

Fig. 4. Impact of Replica Deployment

because the processing cost itself dominates the cost of using more stream replicas.

E. Impact of Replica Deployment

Fig. 4 shows the impact of replica deployment using results from our simulator. For the results, we ran the simulator for a month in simulation time. Then, we plotted the latency distribution for all the output tuples that appeared at 10 different applications. In both figures, $k_{\max}$ represents the degree of replication. The ratios within parentheses represent the relative bandwidth usage and network cost, respectively, compared to those in the non-replication case (i.e., $k_{\max}$=1). In Fig. 4(a), $k_{\max} = 4\ (11.37\text{x}, 22.10\text{x})$ illustrates the case where we replicated each operator at 4 random places and, as a result, consumed 11.37 times higher bandwidth and incurred 22.10 times higher network cost.

In all of the cases, the relative bandwidth usage was less than $k_{\max}^2$. This is because there were (1) $k_{\max}^2$ stream replicas between $k_{\max}$ upstream operator replicas and $k_{\max}$ downstream operator replicas and (2) $k_{\max}$ stream replicas between a stream source and $k_{\max}$ downstream replicas, and between $k_{\max}$ upstream replicas and an application.

As defined in Section V-A, the network cost is a bandwidth-delay product. Because we started from an optimal, non-replicated deployment, stream replicas added later had longer delays than the previous ones. For this reason, the network cost ratio is usually higher than the bandwidth ratio.

Fig. 4 shows that deploying replicas using Algorithm 5 (labeled "min-cost") provides a better latency guarantee than deploying replicas at random servers (labeled "random"). In the random deployment case, there were latencies beyond 16 seconds even though a 13.39 times higher network cost was paid. In the min-cost case, the latency was always smaller than 1 second for a smaller network cost (10.99x). This is because our deployment algorithm finds, among the servers that are likely to achieve the desired availability level, those that minimally increase the network cost.

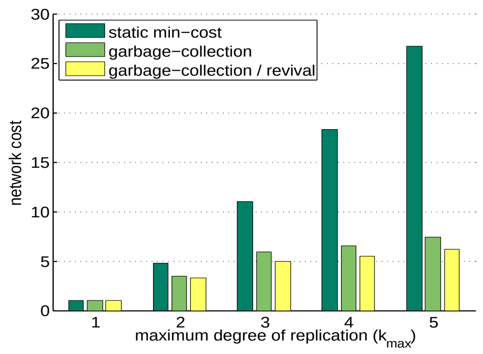


Fig. 5. Impact of Garbage Collection

F. Impact of Garbage Collection

As described in Section V-B, keeping all stream replicas may waste resources without any gain in performance and reliability. In this experiment, for each $k_{\max}$ value, we first achieved the latency guarantee in Fig. 4(b) by running Algorithm 5. In Fig. 5, the first bar for each $k_{\max}$ value represents the network cost in this case (labeled "static min-cost"). Then, we tested how much network cost could be saved through garbage collection without degrading the latency guarantee. The second bar for each $k_{\max}$ value represents the network cost after garbage collection. Thus, the difference between the first two bars in each case represents the network cost of the streams that did not contribute to performance and reliability. Fig. 5 also shows that, as the degree of replication increases, only a smaller portion of stream replicas are useful. We also tested the case where replicas react to changes in system conditions as described in V-C (labeled "garbage-collection / revival"). In this case, we can more aggressively garbage-collect replicas because those garbage-collected can be reused whenever necessary.

VII. Related Work

Techniques for highly-available stream processing so far deploy k peer replicas on independent servers to mask (k-1) simultaneous failures. Most of them are based on the failover model where each operator replica receives data from only one of the upstream replicas and, if the upstream replica fails, the downstream replica must switch to another functioning upstream replica to continue processing [11], [9], [12], [10].

Approaches in [11], [12] and active standby in [9] execute all replicas in parallel. In these approaches, all the replicas are "up-to-date". Thus, a failure stalls the processing only during the failover. The resource usage in these approaches increases in proportion to the degree of replication.

In contrast to the "active replicas" approaches, passive standby and upstream backup [9] combine active and passive replicas to reduce the system usage. In passive standby, active replicas periodically copy the change in their states to passive replicas. In upstream backup, active replicas log outputs so that, if a downstream replica fails, an empty passive replica can use the logged data to rebuild the latest state of the failed one. These passive backup techniques have a slower recovery speed than the "active replicas" approaches because a passive replica, after failover, must bring its old state up-to-date by redoing the recent computation. An approach tackles this problem by distributed checkpointing and parallel recovery [10].

All the failover techniques above use less resources than our approach because at most one of the peer replicas participates in each data flow. Furthermore, the down time after a failure depends on the failover speed. Therefore, they are well-suited in environments with reliable communication and limited resources. In this paper, we assume scenarios where events occurring around the world must be processed in near real time over a large shared substrate. In such scenarios, our approach has distinct advantages because it always benefits from the best of many independent data flows.

Recently, Murty and Welsh presented a high-level vision of a dependable architecture for Internet-scale sensing [19]. They proposed a technique that allows replicas to arbitrarily diverge and then reconciles results from such replicas by finding a representative value (such as the median). In contrast, our approach prevents any side-effects of replication, while striving to give as much freedom to each replica as possible.

Deploying operators in the non-replication context was studied in [17], [18]. Similar to this work, our approach finds a resource-efficient deployment. Our approach, however, considers other aspects (such as the risk of peer replicas' falling into the same network partition) to accomplish the desired availability level. Our approach also garbage-collects/revives replicas to cope with the dynamics of the environment. We presented a preliminary design of our work in [20].

VIII. Conclusion

Today's applications often require service level agreements (SLAs) on the latency of results. If the network is unable to deliver results within these SLAs, we can consider this as a failure. In this paper, we introduce a replication-based approach that can cope with both fail-stop failures and unacceptable latencies. The central notion behind the approach is to replicate operators and let them flow outputs downstream in parallel. In this way, any replica in the system can use whichever data arrives first from upstream replicas. Therefore, the system naturally achieves low-latency processing as well as robustness against server and network problems.

According to this replication framework, we also devise processing primitives that, despite the complications introduced by replication, can provide the same semantic guarantee as those devised for non-replication scenarios. In particular, these primitives allow running replicas differently to avoid the overhead of previous approaches. They also merge stream replicas into a non-duplicate stream. If such a stream feeds order-sensitive operators/applications, our primitives can sort the stream to restore the order that would appear in the non-replication scenario.

Another contribution made in this paper is a strategy for managing replicas at distant servers. Our strategy strives to achieve the best latency guarantee, relative to the cost of replication, while coping with changes in system conditions.

Finally, we present results obtained from our prototype as well as a detailed simulator. These results demonstrate how our approach can overcome the limitations of previous approaches in wide area networks. They also show that, when resources allow, our replication technique is both feasible and correct.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[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*, 2001, pp. 97–108.

[8] V. Paxon, "End-to-end routing behavior in the internet," *IEEE ACM Transactions on Networking*, vol. 5, no. 5, pp. 601–615, 1997.

[9] 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.

[10] 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.

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

[12] 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.

[13] "http://www.planet-lab.org."

[14] 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.

[15] http://www.ntp.org.

[16] P. Tucker, D. Maier, T. Shreard, and L. Fegaras, "Exploiting punctuation semantics in continuous data streams," *IEEE TKDE*, vol. 15, no. 3, pp. 555–568, 2003.

[17] Y. Ahmad and U. Cetintemel, "Networked query processing for distributed stream-based applications," *VLDB*, 2004.

[18] P. Peitzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer, "Network-aware operator palcement for stream-processing systems," *ICDE*, 2006.

[19] 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.

[20] J.-H. Hwang, U. Çetintemel, and S. Zdonik, "Fast and reliable stream processing over wide area networks," in *IEEE SSPS Workshop*, 2007.