A Scalable Failure Recovery Model for Tree-based Overlay Networks

Dorian C. Arnold
Department of Computer Science
University of New Mexico
darnold@cs.unm.edu

Barton P. Miller
Computer Sciences Department
University of Wisconsin
bart@cs.wisc.edu

ABSTRACT

Tree-based overlay networks (TBŌNs) exploit the logarithmic scaling property of trees to provide scalable data multicast, gather, and aggregation services. We leverage the characteristics of many TBŌN computations to develop a new failure recovery model, *state compensation*. State compensation uses: (1) inherently redundant information from surviving processes to compensate for information lost due to failures, (2) weak data consistency to simplify recovery mechanisms, and (3) localized protocols for autonomous process failure recovery. State compensation incurs no overhead in the absence of failures, and when failures occur, only a small number of processes participate in recovery.

We use a formal specification of our data aggregation model to validate state compensation and identify its requirements and limitations. Generally, state compensation requires that data aggregation operations be commutative and associative. We describe an implementation of one state compensation mechanisms and evaluate its recovery performance: for a TBŌN that can support millions of application processes, state compensation can yield millisecond failure recovery latencies and inconsequential application perturbation.

1. INTRODUCTION

Reliable computing models have become critical as the sizes and complexities of HPC systems continue to increase. Currently, there exists at least four systems with more than 100,000 processors and imminent million-processor systems. Expected failure rates at these scales have raised serious concerns over current fault tolerance solutions [9, 10].

Meanwhile, tree-based overlay networks (TBŌNs), hierarchical networks of processes that leverage the logarithmic scaling property of the tree organization, have become a common solution for data aggregation in such large scale environments. As early as 1980, Ladner and Fischer used hierarchical decomposition for efficient, parallel prefix computations [14]. Today, TBŌNs are used in many software systems for a variety of data aggregation operations. For example, Ganglia [24] uses *monitoring trees* to compute summary statistics (sums, averages, minimums, maximums) of node information. Ygdrasil [6] uses *aggregator trees* to condense (nearly) identical text from tools like debuggers. TAG [15] uses an SQL interface to query and aggregate sensor network data, and MRNet has been used for clock synchronization, equivalence classification, time-aligned data aggregation, performance analysis, and debugging [3, 18, 22, 23].

In this paper, we describe *state compensation*, a novel failure recovery model for large TBŌNs with high throughput, low latency aggregation requirements. The key features are:

1. no resource overhead during normal execution;
2. failure recovery using a small number of independently acting processes, and
3. broad applicability to many TBŌN computations.

State compensation uses inherently redundant state from processes that survive failures to compensate for lost information, thereby avoiding explicit state replication. This inherent redundancy is found in many *stateful* TBŌN computations in which processes use state to carry side effects from one aggregation instantiation to another. As information is propagated from the TBŌN leaves to its root, aggregation state, which generally encapsulates the history of processed information, is replicated at successive levels in the tree. We use weak data consistency models to simplify recovery protocols, like tree reconfiguration and information dissemination. We also use localized protocols that allow TBŌN processes to recover independently. Our results show that a TBŌN with millions of application processes can yield millisecond recovery latencies and inconsequential application perturbation.

After discussing related research in Section 2, we specify our computational model and its fundamental properties in Sections 3 and 4. In Section 5, we use our TBŌN specification to describe and validate theoretically the correctness of the state compensation model. We present an MRNet-based implementation of state compensation in Section 6 and our performance evaluation in Section 7. We conclude with a discussion of open issues and future work.

2. RELATED WORK

Related fault tolerance mechanisms can be categorized as *hot backup*, *rollback-recovery*, or *reliable data aggregation*. In fail-over based schemes [1, 25], hosts or processes periodically synchronize their states with backup replicas used to replace failed primary components. Hot backup is general and yields

low recovery latencies since backups are kept in (near) ready states. However, synchronization and resource utilization during normal operation limit scalability. For example, with one backup per primary, hot backups incur a 100% overhead.

In rollback recovery, processes periodically checkpoint their state to persistent storage. Upon failures, the system recovers to the state of the most recent checkpoint [8]. In coordinated checkpointing, the common variant of distributed rollback recovery, processes coordinate to record a globally consistent checkpoint. Rollback recovery is a well studied, general fault tolerance mechanism. However, recent studies [9, 10] predict poor utilization for applications running on imminent systems and the need for resources dedicated to reliability.

Reliable data aggregation has been explored in stream processing engines (SPEs), distributed information management systems (DIMS) and mobile ad hoc networks (MANETs). For reliable SPEs [5, 12], hot backups have been used to replicate query processing nodes; these systems bear the advantages and disadvantages of hot backups discussed above.

Like TBŌNs, DIMS often use hierarchical structures for scalable aggregation. In Astrolabe [20] and the aggregation approach proposed by Gupta et al. [11], nodes are organized into disjoint clusters, and a hierarchy is imposed by forming larger clusters of clusters. Data are propagated within and across the clusters using periodic, random *gossiping* [21]. Gossip-based protocols are scalable with a configurable tradeoff between overhead and robustness. However, they are best suited for applications with small data sets (hundreds to thousands of bytes) that do not require low latency communication and can tolerate partial, non-deterministic output.

SDIMS [26] organizes nodes into a tree and uses an explicit replication protocol in which nodes scatter attribute information (in raw and summary form) to their ascendants and descendants. This approach incurs potentially high replication overhead as data are replicated in multiple places.

Generally, for robust MANET data aggregation [13, 16, 17, 19], nodes use unreliable transport protocols to disseminate locally known attribute data periodically. Nodes merge received attribute data with their local data such that as more partial data is received, local aggregate estimates converge to their actual values. These protocols generally exhibit good scalability characteristics and convergence rates. However, proposed protocols have been specific to relatively simple aggregation operations like sum, max, and average.

We provide a scalable solution for TBŌN computations with deterministic, high throughput delivery without message sizes constraints. We exploit the inherent redundancies of certain classes of TBŌN computations, avoiding any explicit data replication or overhead during normal operation. We are unaware of any other fault tolerance approach that does not rely on explicit replication. Also, recovery is fast, involves a small subset of the TBŌN and yields little perturbation.

3. THE TBŌN MODEL

The TBŌN aggregation model is based on the functional decomposition technique, divide and conquer. The aggregation is decomposed by iteratively sub-dividing its inputs. The

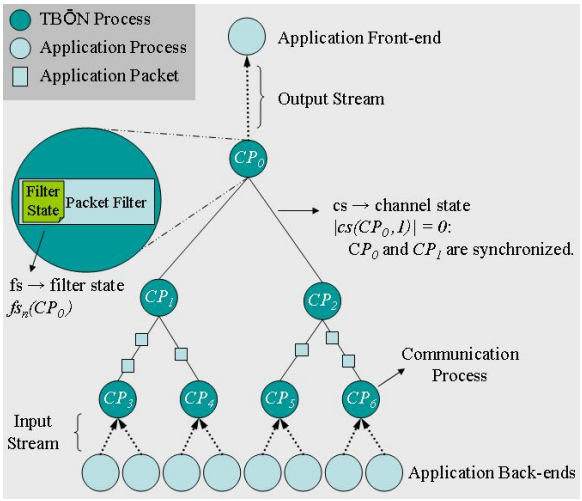

Figure 1: The TBŌN Model: back-ends stream data into the TBŌN; communication processes filter this data and propagate results to the front-end.

TBŌN then maps the sub-problems to different computational resources to compute their solutions efficiently.

3.1 Computational Specification

Figure 1 shows TBŌN processes aggregating and propagating a continuous dataflow from application[1] back-end processes at the leaves to the application front-end at the root. For simplicity, we illustrate balanced, binary trees; the model only requires fully connected trees. Collectively, the front-end and back-end processes are called *end-points*, and the TBŌN's root, leaf, and internal processes are called *communication processes*. Communication processes, denoted by CP_i, where i is a unique identifier, transmit data packets to each other via a reliable, order-preserving transport mechanism, like TCP. Front-ends use *streams* to multicast data to and gather data from groups of back-ends. A stream specifies the end-points participating in a logical dataflow and distinguishes packets from different dataflows. A stream also specifies the aggregation to be applied to the stream's packets.

Parent processes have a set of input channels, one per child, upon which they receive input packets: $in_n(CP_i, j)$ specifies CP_i's n^{th} input from its j^{th} channel. Child processes have output channels used to propagate output packets to their parents: $out_n(CP_i)$ is CP_i's n^{th} output packet. A child's outputs eventually become its parent's inputs:

$$out_n(CP_j) = in_n(CP_i, l) \quad (1)$$

where CP_j is the source for CP_i's l^{th} channel.

A *channel's state* is its incident vector of in-transit packets: $cs_{m,n}(CP_i, j)$ is the vector of in-transit packets to CP_i on

[1]We use "application" to mean the system directly using the TBŌN, beit a software tool or an actual application.

its j^{th} channel when CP_i has received m packets from this channel, and the channel's source has sent n packets, $m \leq n$:

$$cs_{m,n}(CP_i, j) = [in_{m+1}(CP_i, j), \ \ldots \ , in_n(CP_i, j)] \quad (2)$$

$cs(CP_i)$ is the state incident on all CP_i's input channels:

$$cs(CP_i) = \bigsqcup_{j=0}^{fanout(CP_i)-1} cs(CP_i, j) \quad (3)$$

where $fanout(CP_i)$ is the number of CP_i's input channels.

3.2 Data Aggregation

TBŌN processes use *filters* to aggregate input data packets from their children. A filter executes when an input from every channel is available and produces a single output. We call this complete vector of inputs a *wave*; $in_n(CP_i)$ designates CP_i's n^{th} input wave.

Our computational model is based on *stateful* filters with time variant state size. Such filters use *filter state* to carry side effects from one invocation to the next, and the size of this state can become large. However, we can leverage filter state, which encapsulates or summarizes previously filtered inputs, to propagate incremental updates efficiently. For example, consider the *sub-graph folding* filter [23], which continuously merges input sub-graphs into a single graph. Each communication process stores as its state the current merged graph, which encapsulates that process' history of filtered sub-graphs. As new sub-graphs arrive, the filter only needs to output incremental changes to its current merged graph. Using $fs_n(CP_i)$ to represent CP_i's filter state after it has filtered n input waves, we define a filter function, f:

$$f(in_n(CP_i), fs_n(CP_i)) \rightarrow \{out_n(CP_i), fs_{n+1}(CP_i)\} \quad (4)$$

That is, a filter function inputs a wave of packets and its current filter state and outputs a single (potentially null-valued) packet while updating its local state. There can be multiple active streams each with its own filter instance. Generally, the filter function can be abstracted into two operations: a join operation, $\sqcup$, which merges new inputs and filter states, and a difference operation, $-$, which computes the incremental difference between two states.

State join. The join operator, $\sqcup$, merges inputs to comprise a wave:

$$in_n(CP_i) = \bigsqcup_{j:0}^{fanout(CP_i)-1} in_n(CP_i, j),$$

where $fanout(CP_i)$ is CP_i's fan-out. A filter also uses the join operator to update its state with these input waves:

$$in_n(CP_i) \sqcup fs_n(CP_i) \rightarrow fs_{n+1}(CP_i) \quad (5)$$

Deductively, a process' filter state is the join of its filtered inputs: after CP_i has filtered n waves of input,

$$fs_n(CP_i) = in_0(CP_i) \sqcup \ldots \sqcup in_{n-1}(CP_i). \quad (6)$$

We assume that the join operator has these properties:

$$\textbf{A}\text{ssociativity}: \quad (a \sqcup b) \sqcup c = a \sqcup (b \sqcup c)$$
$$\textbf{C}\text{ommutativity}: \quad a \sqcup b = b \sqcup a$$

For the recovery mechanism we present in this paper, we also assume that the join operator is idempotent, $\forall x, x \sqcup x = x$. Many stateful aggregation operations, including the majority of the existing MRNet-based data aggregation operations [3, 4, 22, 23], are idempotent. Specific algorithms classes (all complicit with our recovery model) include set union, graph folding, equivalence class computations, and upper and lower bounds computations. Variations of these idempotent operations that include membership statistics, for example set union with membership counts, are non-idempotent. The relevance of commutativity and associativity has been noted previously in parallel prefix computations [14]. These properties simplify our recovery mechanisms; for example, since correctness does not depend on the grouping and ordering of input data, disconnected sub-trees may reconnect to any branch of the main tree.

State difference. Filter functions based on idempotent joins may output either incremental or complete updates. For efficiency, we favor outputting incremental updates:

$$out_n(CP_i) = fs_{n+1}(CP_i) - fs_n(CP_i) \quad (7)$$

However, as we see in Section 5.3, the option to send complete updates simplify our recovery protocol. For idempotent joins, we describe "$-$" as the *contextual inverse* of $\sqcup$. A function f is contextually invertible if $f(x_1, x_2) \rightarrow y$, and there exists f^{-1} such that:

$$f^{-1}(y, x_1) \rightarrow x_3 : f(x_1, x_2) = f(x_1, x_3), \text{ and}$$
$$f^{-1}(y, x_2) \rightarrow x_4 : f(x_1, x_2) = f(x_4, x_2),$$

where x_2 is not necessarily equal to x_3, but for f, $x2 \equiv x3$ in the context of x_1; likewise for x_1 and x_4. Consider the example where $\sqcup$ is set union, and $-$ is set difference: $\{1,2,3\} \sqcup \{2,3,4\} = \{1,2,3,4\}$, but $\{1,2,3,4\} - \{1,2,3\} = \{4\} \neq \{2,3,4\}$; however $\{1,2,3\} \sqcup \{2,3,4\} = \{1,2,3\} \sqcup \{4\}$.

3.3 A TBŌN Example

As an example of the TBŌN data aggregation model at work, consider an integer union computation. Integer data are propagated through the TBŌN from the leaves to the root. Each process filters its input stream to suppress duplicates and send the unique values to its parent. The persistent state at each process is that process' set of filtered integers; the state of each channel is its incident vector of pending incremental updates from the child of the channel to its parent. The final output at the application front-end is the overall set of unique integers sent by the back-ends. In this example, '$\sqcup$' is set union, '$-$' is set difference. While this example is simple, it is representative of many more complex aggregations. For example, graph merging, used in the sub-graph folding algorithm [23] of the Paradyn tool and the stack trace analysis of the STAT tool [3], is essentially the same computation on graph data. That is, the TBŌN performs union and difference operations on node and edge data instead of integers.

4. FUNDAMENTAL TBŌN PROPERTIES

Our goal is to develop scalable TBŌN recovery mechanisms by avoiding the explicit data replication that leads to non-scalable resource consumption. Our solution, state compensation, is based on the TBŌN characteristics described in Section 3 and motivated primarily by three properties:

1. *The Inherent Redundancy Lemma*: inherent information redundancies exist in stateful TBŌN-based data aggregations. Intuitively, as data is propagated from the leaves toward the root, aggregation state, which generally encapsulates the history of processed inputs, is replicated at successive levels in the tree.
2. *The TBŌN Output Dependence Lemma*: the output of a TBŌN subtree depends upon solely the filter state at the subtree's root and the subtree's channel states. Intuitively, in-flight data triggers the execution of data aggregations, the output of which depends upon the inputs and the current state of the filtering process.
3. *The All-encompassing Leaf States Lemma*: the states at a TBŌN's leaves contain all the information in the rest of the TBŌN's filter and channel states. Intuitively, since the state of a TBŌN process encapsulates its history of filtered inputs and since leaf processes filter data before any other TBŌN process, the leaves' input histories are the most complete.

When a TBŌN process fails, the filter and channel states associated with that process are lost. Based on these lemmas, we will demonstrate that proper failure recovery only requires the recovery of the channel state. Furthermore, this state can be recovered from the redundant information maintained by processes nearer to the leaves of the TBŌN.

4.1 Inherent Redundancy

The Inherent Redundancy Lemma builds upon the equivalence of the input of a data aggregation operation and its output. Intuitively, the output of a data aggregation is a summarized or compressed form of its input, but the input and the output should contain equivalent information.

LEMMA 4.1 (INPUT/OUTPUT EQUIVALENCE). *TBŌN data aggregation output is equivalent to its given input.*

Proof

(Eqn. 5: $in \sqcup fs = fs'$):
$in_n(CP_p) \sqcup fs_n(CP_p) \quad = \quad fs_{n+1}(CP_p)$

('$-$' is the contextual inverse of '$\sqcup$'):
$in_n(CP_p) \quad \equiv \quad fs_{n+1}(CP_p) - fs_n(CP_p)$

(Eqn. 7: fs' - fs = output):
$\equiv \quad out_n(CP_p)$

■

Now we can demonstrate the inherent TBŌN redundancies:

LEMMA 4.2 (INHERENT REDUNDANCY). *For any parent process, the join of its filter and pending channel states equals the join of its children's states:*

$$\forall CP_i, fs_m(CP_i) \sqcup cs_{m,n}(CP_i, 0) \sqcup cs_{m,p}(CP_i, 1) = fs_n(CP_j) \sqcup fs_p(CP_k),$$

where, as in Figure 2, CP_i has two children[2]*, CP_j and CP_k on input channels 0 and 1, respectively, and CP_i, CP_j, and CP_k have filtered m, n, and p waves of input, respectively; $m \leq n, m \leq p$.*

[2]Our proofs have trivial extensions for arbitrary fan-outs.

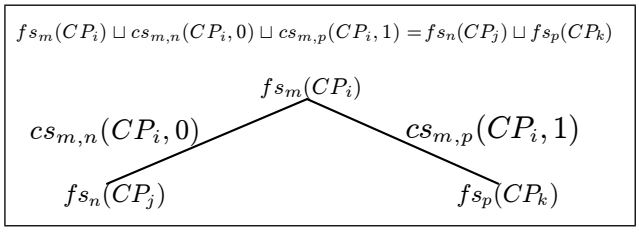

Figure 2: Inherent TBŌN Information Redundancy: the join of a parent's filter and pending channel states equals the join of its children's states.

Proof

$fs_m(CP_i) \sqcup cs_{m,n}(CP_i, 0) \sqcup cs_{m,p}(CP_i, 1)$

(Eqn. 6: filter state = join of input history):
$= \quad in_0(CP_i) \sqcup \ldots \sqcup in_{m-1}(CP_i) \sqcup$
$cs_{m,n}(CP_i, 0) \sqcup cs_{m,p}(CP_i, 1)$

(Eqn. 1: input = join of children's output):
$= \quad (out_0(CP_j) \sqcup out_0(CP_k)) \sqcup \ldots \sqcup$
$(out_{m-1}(CP_j) \sqcup out_{m-1}(CP_k)) \sqcup$
$cs_{m,n}(CP_i, 0) \sqcup cs_{m,p}(CP_i, 1)$

(Eqns. 2 & 1: channel state = channel source output):
$= \quad (out_0(CP_j) \sqcup out_0(CP_k)) \sqcup \ldots \sqcup$
$(out_{m-1}(CP_j) \sqcup out_{m-1}(CP_k)) \sqcup$
$out_m(CP_j) \sqcup \ldots \sqcup (out_n(CP_j) \sqcup$
$out_m(CP_k) \sqcup \ldots \sqcup (out_p(CP_k)$

(Commuting the operands):
$= \quad out_0(CP_j) \sqcup \ldots \sqcup out_n(CP_j) \sqcup$
$out_0(CP_k) \sqcup \ldots \sqcup \ out_p(CP_k)$

(Lem. 4.1: output $\equiv$ input):
$= \quad in_0(CP_j) \ \sqcup \ldots \sqcup in_n(CP_j) \sqcup$
$= \quad in_0(CP_k) \sqcup \ldots \sqcup in_p(CP_k)$

(Eqn. 6: input history = filter state):
$= \quad fs_n(CP_j) \sqcup fs_p(CP_k)$

■

4.2 All-encompassing Leaf States

We now show that the states at a subtree's leaf processes contain all the information available in the rest of that subtree. That is, should any non-leaf channel or filter state be lost, the information needed to regenerate that state exists at the leaves of any subtree that totally contains the lost components. To aid our discussion, we introduce a new operator, $desc^k$, which describes a process' descendants k levels away. For example, $desc^1(CP_i)$ is the set of CP_i's children, and $desc^2(CP_i)$ is the set of CP_i's grandchildren.

The fs and cs operators without subscripts are shorthand for the specified process' or channel's current state. Similarly, without subscripts, *in* and *out* designate the specified process'

input and output history, respectively. Lastly, when these operators are applied to a set of processes or channels, they return the join of that operator applied to the set's members.

LEMMA 4.3 (THE ALL-ENCOMPASSING LEAF STATES). *The join of a subtree's leaf states equals the join of the state at the subtree's root process and the TBŌN in-flight data.*

Proof

From Lemma 4.2, we deduce:

$$fs(desc^1(CP_0)) = fs(desc^0(CP_0)) \sqcup cs(desc^0(CP_0))$$
$$fs(desc^2(CP_0)) = fs(desc^1(CP_0)) \sqcup cs(desc^1(CP_0))$$
$$\ldots$$
$$fs(desc^k(CP_0)) = fs(desc^{k-1}(CP_0)) \sqcup cs(desc^{k-1}(CP_0))$$

Substituting the former identities into the latter, we get:

$$fs(desc^k(CP_0)) = fs(CP_0) \sqcup cs(desc^0(CP_0)) \sqcup \ldots \sqcup cs(desc^{k-1}(CP_0))$$

■

4.3 TBŌN Output Dependence

Finally, we show that the TBŌN computation's output stream is solely a function of the root's filter state and the TBŌN's channel states. The TBŌN input is the stream of inputs filtered by the TBŌN leaves, $in(desc^k(CP_0))$, where k is the TBŌN depth. The effective TBŌN output, $out(CP_0)$, is the stream of outputs produced by the root process if messages channels are flushed and all communication processes become synchronized; that is, the root and the leaf processes have filtered the same number of input waves.

Lemma 4.1 shows the equivalence between the inputs and output of an aggregation operation: $in(CP_i) \equiv out(CP_i)$. We can generalize this to show that the join of the inputs of any level of TBŌN processes are equivalent to the join of the outputs produced by those processes: $in(desc^k(CP_0)) \equiv out(desc^k(CP_0))$. Since output from level k becomes input to level $k-1$, a simple induction yields:

COROLLARY 4.4 (TBŌN INPUT/OUTPUT EQUIVALENCE). *The input to a TBŌN's leaves is equivalent to the effective output of its root:* $in(desc^k(CP_0)) \equiv out(CP_0)$.

We now demonstrate our last fundamental TBŌN properties:

LEMMA 4.5 (TBŌN OUTPUT DEPENDENCE). *The output of a TBŌN computation is solely a function of the root's state and the TBŌN channel states.*

Proof.

(By Cor. 4.4: Input/Output Equivalence)
$out(CP_0) \equiv in(desc^k(CP_0))$

(Eqn. 6: input history = filter state)
$\equiv fs(desc^k(CP_0))$

(By Lem. 4.3: All-encompassing Leaf State)
$\equiv fs(CP_0) \sqcup cs(desc^0(CP_0)) \sqcup \ldots \sqcup cs(desc^{k-1}(CP_0))$

■

No Failures:	7	11	27	35	35
Failures:	7	8	15	35	35
	t_0	t_1	t_2	t_3	Overall Maximum

Figure 3: Convergent Recovery for *integer maximum*. A failure at t_0 causes a divergence in the output at t_1 and t_2; at t_3, the output re-converges.

5. STATE COMPENSATION

In state compensation, we merge states from non-failed TBŌN processes to compensate for lost state. After discussing our failure and data consistency models, we describe *state composition*, our primary compensation mechanism, and use our previous results to prove its correctness.

5.1 Failure Model

We assume *fail-stop* process failures, detectable failures that cause processes to cease to produce new output. Any TBŌN process (root, leaf or internal) may fail at any time, even simultaneously. Should all TBŌN processes fail, the TBŌN degenerates to a one level tree with the front-end directly connected to the back-ends. Failed processes need not be replaced before the system can resume normal operation: the TBŌN is reconfigured to omit failed processes, but new processes (on repaired hosts) may be integrated dynamically into the TBŌN. Network device failures that cause a permanent network partition are treated as failures of the processes partitioned from the TBŌN's root process.

State compensation does not address the failures of application processes (outside the TBŌN). These processes may be viewed as sequential data sources and sinks amenable to sequential checkpointing, which avoids the coordination complexities of distributed checkpointing. This solution does not work if the application uses communication channels external to the TBŌN – we have not yet seen this case.

5.2 Data Consistency Model

State compensation guarantees weak data consistency, called *convergent output recovery* [12], in which failures may cause intermediate TBŌN output data to diverge from the output produced by an equivalent computation with no failures. Relying on associativity, commutativity and idempotence, our recovery mechanisms may cause the associations and commutations of input data that have been re-routed due to failure(s) to differ from that of the non-failed execution, or there may be some duplicate input processing. These phenomena cause the output divergence. Eventually, the output stream converges back to that of the non-failed computation after all input affected by the failures are propagated to the root and it starts to process input not impacted by the failures. Consider Figure 3, which shows the output streams of two originally identical TBŌNs executing an *integer maximum* aggregation: the front-end continuously outputs new maximums as they are filtered. A failure at t_0 causes a divergence in the output at t_1 and t_2. At t_3, the output re-converges to that of the non-failed execution. Convergent recovery preserves all output information and produces no extraneous output.

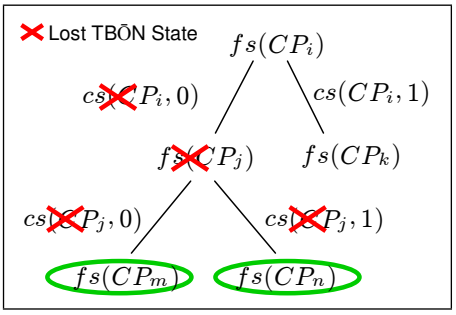

Figure 4: State Composition: When CP_j fails, $fs(CP_j)$, $cs(CP_j)$, and $cs(CP_i, 0)$ are lost. $fs(CP_m)$ and $fs(CP_n)$, can be used to compensate for the loss.

5.3 State Composition

State composition uses TBŌN state from processes below failure zones to compensate for lost state. This strategy is motivated primarily by the All-encompassing Leaf State Lemma, 4.3, which states that for any subtree, the state at the leaves of the subtree subsume the rest of the TBŌN state. As shown in Figure 4, when a TBŌN process fails, the filter and channel's states associated with that process are lost. State composition compensates for this lost state using state from orphaned processes. Specifically, after the orphans are re-adopted into the tree, they propagate their filter state as output to their new parent. We call this *state composition* because the compensating states form a composite equivalent to the state that has been lost.

THEOREM 5.1 (STATE COMPOSITION). *A TBŌN can tolerate failures without changing the computation's semantics by re-introducing filter state from the descendants of failed processes as channel state.*

Proof. Consider the TBŌN in Figure 4. If CP_j fails, the TBŌN loses the following states: $fs(CP_j)$, $cs(CP_i, 0)$, $cs(CP_j, 0)$, and $cs(CP_j, 1)$. By Theorem 4.5, the TBŌN's output only depends upon the system's root and channel states. Therefore, we only need to show that propagating as output the states of CP_j's children compensates for the lost states, $cs(CP_i, 0)$, $cs(CP_j, 0)$ and $cs(CP_j, 1)$. In other words, we show that the composition of the states of the failed CP_j's children subsume the lost channel states. Theorem 4.3 says that for any subtree, the filter states at the leaves subsume the states throughout the rest of this subtree. In this case, CP_m and CP_n's states subsume $cs(CP_i, 0)$, $cs(CP_j, 0)$ and $cs(CP_j, 1)$ and, therefore, can replace those states without changing the computation's semantics. The composition of CP_m and CP_n's states may contain input data already processed by CP_i, so the aggregation operation must be idempotent, that is, resilient to processing the same input data multiple times. ∎

The result of the State Composition Theorem is that for TBŌNs executing idempotent data aggregation operations, we can recover **a**ll information lost due to process failures simply by having orphaned processes transmit their filter state to their new parents. Generally, the time to filter the aggregated states used for compensation is less than the time to filter the original data that constituted the aggregate.

If the TBŌN root fails, we do not know what output has already been received by the application front-end and must act conservatively. We regenerate the entire TBŌN output stream. When the root process fails, one of its children is promoted to the root position, and the remaining orphans become the new root's descendants. In this case, the orphaned processes transmit their filter state directly to the new root, not (necessarily) their new parent. The new root merges these filter states with its own resulting in a composite of the input history of the original root's children. In other words, the composition output subsumes all output (missing or otherwise) that the failed root process could have propagated to the front-end. This output is propagated to the front-end process.

In many situations, application back-ends, which connect to the TBŌN leaf processes, also aggregate data from multiple sources. For example, in the stack trace analysis tool [3], each tool back-end collects and aggregates stack traces from all collocated application processes. Therefore, filters are executed in the application back-ends to aggregate local data. As a result, the back-ends also maintain persistent filter state, which encapsulates the history of inputs propagated by that back-end. Should a TBŌN leaf process fail, we compose the filter states from the orphaned back-end processes once they reconnect to the TBŌN.

State composition can handle multiple failures that overlap in time. When multiple unrelated failures occur, the orphans from these failures simply propagate their compensating filter state to their new parent upon reintegration into the TBŌN. If the adopting parent fails as the orphan attempts reintegration, the orphan finds a new adopter for reintegration. If the orphan fails before it is reintegrated, the orphan's children now become orphaned and initiate their own failure recovery.

5.4 State Decomposition

State composition may over-compensate for lost state by retransmitting some non-lost state and relies on idempotence to compute the aggregation correctly when input data is processed more than once. We also have developed a *state decomposition* mechanism to address non-idempotent computations. State decomposition precisely calculates lost information and compensates for only that information thereby removing any potential for re-processing the same input data multiple times. Intuitively, the failed process' parent (eventually) should filter the same input information as the surviving processes directly below it, namely, the children and siblings of the failed process. Using the filter states (input histories) of the failed process' parent and the failed process' siblings' and children, decomposition precisely computes what input information from the latter has been filtered by the former. A more detailed discussion of decomposition is beyond the scope of this paper.

6. IMPLEMENTATION

The components of our TBŌN failure recovery model are failure detection, tree reconfiguration and lost state compensation. Accordingly, we have implemented a reliable TBŌN by adding an event detection service for failures and other events, a protocol for dynamic topology (re-)configuration and an implementation of state composition to the MRNet [22] TBŌN prototype, as we now describe.

6.1 MRNet Event Detection

Each MRNet process must detect certain asynchronous system events like process failures or adoption requests for reconfigurations. We use a passive, connection-based mechanism (TCP-based in this case) to notify processes of relevant events. The new MRNet event detection service (EDS) runs as a thread within each process and primarily monitors a *watch list* of designated *event sockets*. These sockets include a *listening socket* to which other processes can connect to establish *process peer* relationships. The EDS uses the *select* system call to wait until a specified event occurs on at least one monitored socket.

Failure Detection. Failure detection in a distributed system comprises component failure detection and failure information dissemination. For component failure detection, we leverage the TBŌN structure to establish small groups of processes that monitor each other. Currently, an MRNet process monitors its parent and children using special *failure-detection event sockets*. An error detected on a failure-detection event socket indicates that the process peer has failed. The timeliness of this failure detection mechanism depends on the cause of failure. When a process terminates, its host's kernel will abort the process' open connections; this is detected immediately by the process' peers. However, node and link failures prevent a kernel from explicitly aborting its connections. TCP keep-alive probes can be used to detect such failures, but in general keep-alive probes have a two hour default period that can be lowered only on a system-wide basis by privileged users. User-level heartbeat protocols [2] could provide responsive, user controlled node and link failure detection.

Process failure events must be disseminated to all TBŌN processes so that they may update their topologies. Once again, we leverage the TBŌN structure for efficient information dissemination. Multiple peer EDSes will detect each process failure. An EDS that detects its parent's failure reports that failure to its children, and an EDS that detects that one of its children have failed reports that failure to its parent and surviving children. A failure report contains the rank of the failed process. Upon receiving a failure report, a process propagates the report to all the process' peers other than the peer from which the report was received.

Reconfigurations and propagation delays can lead to duplicate, late, missing or out-of-order failure reports. For example, a process can receive a duplicate failure report if the process has been adopted multiple times to different TBŌN branches and receives the same report from multiple branches. Acting on a failure report is an idempotent operation: a duplicate failure report reiterates that a process has failed. Late or missing failure reports lead to stale topology information. Our reconfiguration algorithm, summarized below, tolerates stale topology information by iteratively connecting to potential adopters, which may have failed, until an adoption succeeds. Different failure reports cannot contain conflicting information, so out-of-order failure reports do not pose any issue beyond that of stale topology information.

Dynamic Topologies. Dynamic topology reconfigurations are necessary to accommodate process failures. After failures, orphans initiate a reconfiguration algorithm to determine a new parent that re-establishes a path to the front-end. Each orphan generates a sorted list of potential adopters, so that should a selected adopter fail by the time an orphan tries to connect to it, the orphan can attempt to connect to a different adopter. While the complete details are beyond the scope of this paper, we analyzed several reconfiguration algorithms that are executed by orphans without coordination. We focused on: (1) the costs of disseminating and managing the TBŌN process information needed by the algorithms, (2) the algorithms' execution times, and (3) the data aggregation latency of the resulting configurations. We observed that tree height increases can have a significant negative impact on data aggregation performance and chose an algorithm that restricts tree height increases and moderately increases fan-out as failures occur.

An orphan contacts its chosen adopter's EDS, and the adopter and adoptee establish a socket for application data transmission. After the adoption, a reconfiguration report, containing the adopter's and adoptee's ranks, must be reported to all processes. Reconfiguration reports are disseminated just as failure reports are: an adoptee sends the reconfiguration report to its children, and the adopter sends the report to its parent and other children. A process that receives a report sends the report to all peers other than the one from which the report was received.

As with failure reports, acting on a reconfiguration report is an idempotent operation: a replicated report reiterates the adoption of a child by its new parent. However, reconfiguration reports of different adoptions regarding the same orphan are conflicting. If processed in the wrong order, topology information will become incorrect. We adopt the concept of *incarnation versions* [7] to address this problem. Each process maintains an incarnation number. After each adoption, an orphan's incarnation number is incremented and propagated with the recovery report. Processes disregard reconfiguration reports regarding orphans for whom they have received a report with a higher incarnation number. Late reconfiguration reports may lead to erroneous cycles in the topology. For example, if two orphans from different, simultaneous failures each select a descendant of the other as its adopter, a cycle is produced. Our current prototype performs a simple topology validation that avoids *most* instances of cycle formation. Late reconfiguration updates also may lead to correct but sub-optimal topologies, for example, if a process with a stale topology wrongly computes that an adoption would not lead to a height increase.

6.2 State Composition Implementation

We derived a straightforward implementation of state composition from the theory from Section 5.3. As shown in Algorithm 6.1, when a child detects its parent's failure, the orphaned child pauses input processing[3], computes a sorted list of potential adopters and establishes a connection with the first surviving process from the list. After reconfiguration,

[3] An orphan could filter new input and buffer output until it has been adopted. In fact, since data aggregation is commutative and associative, upon adoption it would be sufficient for an orphan to propagate the aggregate of its pending output.

Algorithm 6.1: Failure Recovery Algorithm

```
if parent fails then
    Pause input data processing;
    begin TBŌN reconfiguration
        Compute sorted potential adopters list;
        while failed to connect to head of list do
            Remove list head;
    end
    Update network topology data structure;
    begin TBŌN state recovery
        foreach stream do
            Propagate filter state to new parent;
    end
if child fails then
    Update network topology data structure;
Propagate failure and reconfiguration reports;
Resume normal operation;
```

the adoptee and adopter update their topologies to reflect the changes. The adoptee then compensates for any lost state by propagating its filter states to its adopter.

A parent process that detects the failure of one of its children simply deletes the failed process from its topology data structure. After the failure recovery process completes, the parent of the failed process, the adopted processes and their adopters disseminate failure and reconfiguration reports.

7. EVALUATION

In the absence of failures, state compensation consumes no computational resources beyond those necessary for normal TBŌN operation. We evaluated the performance of failure recovery and the impact of failures on application performance. Our experiments were run on the Lawrence Livermore National Laboratory's Atlas Cluster of 1,024 2.4 GHz AMD Opteron nodes, each with 8 CPUs and 16 GB of memory and linked by a double data rate InfiniBand network.

7.1 The Experimental Framework

The main component of our experimental framework was a failure injection and management service (FIMS) that injected process failures and collected recovery performance data. After failure recovery, each formerly orphaned process notified the FIMS that its failure recovery was complete. The FIMS conservatively estimated the overall TBŌN recovery latency using the time of failure injection and the time of receipt of the last recovery completion message. The estimate was conservative since it includes transmission and serialization delays.

7.2 The Application

We used the integer union computation introduced in Section 3.3, which computes the set of unique integers in the TBŌN's input stream by filtering out duplicates, to test our failure recovery mechanisms. The application back-ends propagate randomly generated integers at a rate of ten packets per second, the default stack trace sampling rate of our MRNet-based STAT tool [3]. After each experiment, we compare the input from the back-ends with the output at the front-end. Even when failures are injected, the output set at the front-end must be equal to the union of the input sets. We use the integer union computation because its output is easily verifiable, and, as we discussed, is representative of many useful, more complex aggregations.

7.3 Recovery Latency

When failures occur, the duration of the temporary TBŌN output divergences output can be estimated by:

$$MAX_{i=0}^{norphans}(\ t(recovery(o_i))\ -\ t(failure)\ ) + (\ l(oparent(o_i),\ root\ )\ -\ l(\ nparent(o_i),\ root)\ )$$

where $t(e)$ is the time that event e occurs, $recovery(o_i)$ is the recovery completion of orphan i, $l(src, dst)$ is the propagation latency (possibly over multiple hops) from src to dst, $oparent(o_i)$ is orphan i's old parent, and $nparent(o_i)$ is orphan i's new parent. This formula computes the maximum across all orphans of an orphan's recovery latency and difference in propagation latencies between the orphan's old path to the root and the new path after reconfiguration. Our evaluation is focused on the orphans' recovery latencies, compensation is completed once each orphan has introduced its compensating state as channel state to its new parent. Under our eventual consistency model, diverged output is correct, just not up-to-date. Further, propagation latencies may be shorter after failure, for example, if a failure occurs deep in the tree, and orphans are adopted by the root.

Each orphan's individual failure recovery latency is the sum:

$$l(new\_parent) + l(connect) + l(compensate) + l(cleanup)$$

where $l(new\_parent)$ is the time to compute the new parent, $l(connect)$ is the time to connect to the new parent, $l(compensate)$ is the time to send the filter state[4], and $l(cleanup)$ is the time to update local data structures and propagate failure and reconfiguration reports.

For state composition, only orphaned processes (and adopting parents) participate in failure recovery. Therefore, the failure recovery performance is a function of the tree's fan-out, not total size. Our first experiments evaluated the impact that the number of orphans caused by a failure has on failure recovery latencies. Our MRNet experiences suggested that typical fan-outs range from 16 to 32; however, we tested extreme fan-outs up to 128 since hardware constraints can force such situations. For instance, LLNL's BlueGene/L enforces a 1:128 fan-out from its I/O nodes to its compute nodes. To test such large fan-outs, we organized the micro-benchmark topologies such that only the designated victim processes had the large test fan-out, as shown in Figure 5. We added 16 additional processes to distribute the orphan adoptions; this reflects practical TBŌN topologies in which orphans have multiple potential adopters from which to choose.

For each experiment, we report the FIMS' conservative estimate of the overall TBŌN recovery latency, the maximum individual orphan recovery latency and the average recovery latencies for all orphans. The results are shown in Figure 6. $l(new\_parent)$ and $l(connect)$ dominate the orphans' individual failure recovery latencies. As the number of orphans

[4]Technically, $l(compensate)$ is the latency of the local TCP *send* operation after which it is guaranteed only that the kernel has accepted the compensation data for transmission.

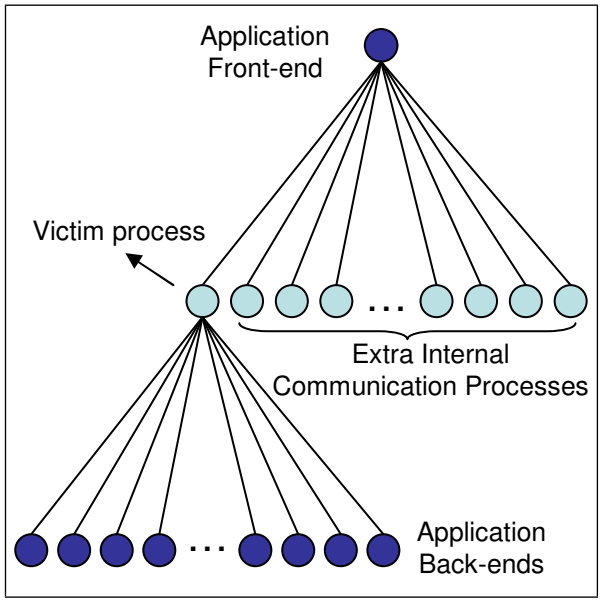

Figure 5: Micro-benchmark Topology: The victim has the fan-out being evaluated, and 16 internal processes are added to distribute orphan adoptions.

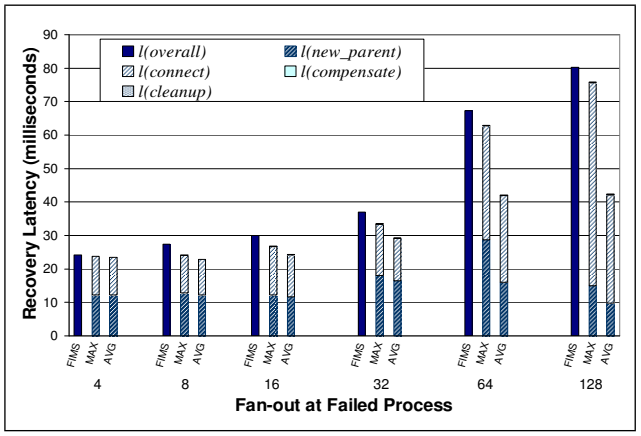

Figure 6: Recovery Latencies: $l(overall)$ is overall TBŌN recovery estimate, $l(new\_parent)$, $l(connect)$, $l(compensate)$ and $l(cleanup)$ are the latencies to choose new parent, connect to parent, transmit filter state, and update local data structures.

increases, an increase in the connection time causes the individual orphan recovery latencies to increase. The increase in connection time can be attributed to serialization at the adopters, since more orphans are being adopted by the same number of adopters. In practical scenarios with more balanced topologies, better adopter/adoptee ratios would mitigate this contention. $l(new\_parent)$ remains relatively constant – the peak in $l(new\_parent)$ for the slowest orphan in the "64 orphans" experiment is an outlier, since the average across the 64 orphans matches those of the other experiments. For larger trees with more processes, $l(new\_parent)$ will increase, but based on additional tree reconfiguration experiments, we have determined that even for a tree of over 10^6 processes, the time to compute a new parent should remain in the hundreds of milliseconds. The major observation of these results is that even considering the FIMS' estimate of overall recovery latency, the latency for our largest fan-out of 128 is less than 80 milliseconds – an insignificant interruption, considering that a 128^3 tree has over 2 million leaves.

7.4 Application Perturbation

We evaluated the impact of failures on application performance by dynamically monitoring the throughput of the integer union computation as we injected TBŌN failures. The experiment started with a 2-level topology and a uniform fan-out of 32. We injected a random failure every 30 seconds killing four of the 32 internal processes. At the application front-end, we tracked the application's throughput reported as the average throughput over the ten most recent output packets. The results in Figure 7 show some occasional dips (and proceeding bursts) in packet arrival rates. There are several dips that do not coincide with the 30, 60, 90 and 120 second marks (indicated by the arrows) at which failure were injected, and some even occur before the first failure was injected. We conclude that these are due to other artifacts, like operating system thread scheduling, and that there is no perceivable change in application's performance due to the injected failures. We are running additional experiments to determine how increases in application data rate will impact these results; however, with our millisecond recovery latencies we still expect little performance perturbation.

8. CONCLUSION

We developed a scalable recovery model for high performance data aggregation that exploits the inherent information redundancies found in many TBŌN computations. To the best of our knowledge, this is the first fault-tolerance research to leverage implicit state replication and avoid the overhead of explicit replication. Furthermore, our recovery protocol is completely distributed. Our state composition mechanism requires only that data aggregations are commutative, associative and idempotent and is suitable for many useful algorithms. Our evaluation shows that we can recover from failures in TBŌNs that can support millions of processes in milliseconds with inconsequential application perturbation.

As HPC system sizes and failure rates continue to increase, no (and low) overhead recovery models like state compensation become essential. We are considering several strategies for extending this work, for example, to accommodate compositions of heterogeneous filter functions, application back-end failures, and even directed acyclic graph topologies. Also, many of our current aggregations are motivated by the analysis requirements of parallel and distributed system tools. Open research questions are how will the characteristics of new TBŌN applications map to the requirements of our failure recovery model and how can we extend the failure recovery model to accommodate applications that do not comply with the model's current requirements.

9. REFERENCES

[1] P. A. Alsberg and J. D. Day. A principle for resilient sharing of distributed resources. In *2nd International Conference on Software Engineering (ICSE '76)*, pages 562–570, San Francisco, CA, 1976. IEEE Computer Society Press.

[2] G. R. Andrews. Paradigms for process interaction in distributed programs. *ACM Computing Surveys*, 23:49–90,

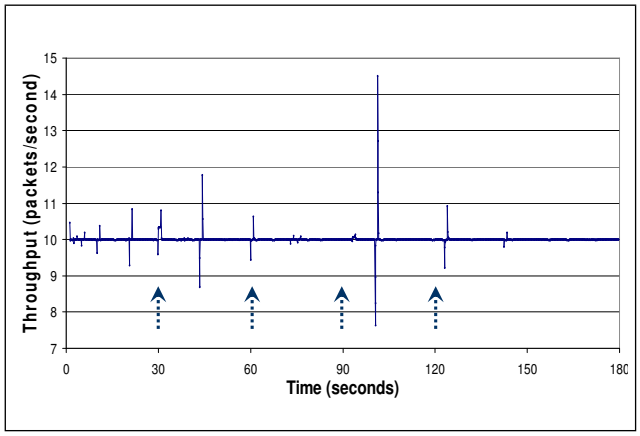

Figure 7: Application Perturbation: Failures (indicated by arrows) are injected every 30 seconds into a TBŌN with an initial 32^3 topology.

1991.

[3] D. C. Arnold, D. H. Ahn, B. R. de Supinski, G. Lee, B. P. Miller, and M. Schulz. Stack trace analysis for large scale applications. In *21st IEEE International Parallel & Distributed Processing Symposium (IPDPS '07)*, Long Beach, CA, March 2007.

[4] D. C. Arnold, G. D. Pack, and B. P. Miller. Tree-based computing for scalable applications. In *11th International Workshop on High-Level Parallel Programming Models and Supportive Environments*, Rhodes, Greece, April 2006.

[5] M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker. Fault-tolerance in the borealis distributed stream processing system. In *SIGMOD International Conference on Management of Data*, pages 13–24, Baltimore, MD, June 2005.

[6] S. M. Balle, J. Bishop, D. LaFrance-Linden, and H. Rifkin. *Applied Parallel Computing*, volume 3732/2006 of *Lecture Notes in Computer Science*, chapter 2, pages 207–216. Springer, February 2006.

[7] A. D. Birrell, R. Levin, M. D. Schroeder, and R. M. Needham. Grapevine: An exercise in distributed computing. *Communications of the ACM*, 25(4):260–274, 1982.

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

[9] E. N. Elnozahy and J. S. Plank. Checkpointing for peta-scale systems: A look into the future of practical rollback-recovery. *IEEE Transactions on Dependable and Secure Computing*, 1(2):97–108, April-June 2004.

[10] G. Gibson, B. Schroeder, and J. Digney. Failure tolerance in petascale computers. *CTWatch Quarterly*, 3(4), November 2007.

[11] I. Gupta. *Building Scalable Solutions to Distributed Computing Problems Using Probalistic Components*. PhD thesis, Cornell University, August 2003.

[12] J.-H. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, and S. Zdonik. High-availability algorithms for distributed stream processing. In *21st International Conference on Data Engineering (ICDE'05)*, pages 779–790, Tokyo, Japan, April 2005.

[13] H. Jiang and S. Jin. Scalable and robust aggregation techniques for extracting statistical information in sensor networks. In *26th IEEE International Conference on Distributed Computing Systems (ICDCS '06)*, page 69, Lisboa, Portugal, July 2006. IEEE Computer Society.

[14] R. E. Ladner and M. J. Fischer. Parallel prefix computation. *Journal of the ACM*, 27(4):831–838, 1980.

[15] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tag: a tiny aggregation service for ad-hoc sensor networks. In *5th Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, December 2002.

[16] A. Manjhi, S. Nath, and P. B. Gibbons. Tributaries and deltas: Efficient and robust aggregation in sensor network streams. In *ACM SIGMOD International Conference on Management of Data (SIGMOD 2005)*, pages 287–298, Baltimore, MD, June 2005. ACM Press New York, NY, USA.

[17] A. Montresor, M. Jelasity, and O. Babaoglu. Robust aggregation protocols for large-scale overlay networks. In *2004 International Conference on Dependable Systems and Networks (DSN 2004)*, page 19, Palazzo dei Congressi, Florence, Italy, June/July 2004. IEEE Computer Society.

[18] A. Nataraj, A. D. Malony, A. Morris, D. Arnold, and B. Miller. A framework for scalable, parallel performance monitoring using tau and mrnet. In *International Workshop on Scalable Tools for High-End Computing (STHEC 2008)*, Island of Kos, Greece, June 2008.

[19] S. Nath, P. B. Gibbons, S. Seshan, and Z. R. Anderson. Synopsis diffusion for robust aggregation in sensor networks. In *2nd International Conference on Embedded Networked Sensor Systems(SenSys '04)*, pages 250–262, Baltimore, MD, November 2004.

[20] R. V. Renesse, K. P. Birman, and W. Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Transactions on Computer Systems*, 21(2):164–206, 2003.

[21] R. V. Renesse, Y. Minsky, and M. Hayden. A gossip-style failure detection service. In *IFIP International Conference Distributed Systems and Platforms and Open Distributed Processing (Middleware 98)*, pages 55–70, The Lake District, England, September 1998.

[22] P. C. Roth, D. C. Arnold, and B. P. Miller. MRNet: A software-based multicast/reduction network for scalable tools. In *2003 ACM/IEEE conference on Supercomputing (SC '03)*, page 21, Phoenix, AZ, November 2003. IEEE Computer Society.

[23] P. C. Roth and B. P. Miller. On-line automated performance diagnosis on thousands of processes. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '06)*, New York, NY, March 2006.

[24] F. D. Sacerdoti, M. J. Katz, M. L. Massie, and D. E. Culler. Wide area cluster monitoring with ganglia. In *IEEE International Conference on Cluster Computing (CLUSTER 2003)*, pages 289–298, Hong Kong, September 2003.

[25] F. B. Schneider. Byzantine generals in action: Implementing fail-stop processors. *ACM Transactions Computer Systems*, 2(2):145–154, May 1984.

[26] P. Yalagandula and M. Dahlin. A scalable distributed information management system. In *2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM '04)*, pages 379–390, Portland, OR, August/September 2004.