Scalable Failure Recovery for High-performance Data Aggregation

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

Barton P. Miller
Computer Sciences Department
University of Wisconsin
Madison, Wisconsin USA
darnold@cs.unm.edu

***Abstract*—Many high-performance tools, applications and infrastructures, such as Paradyn, STAT, TAU, Ganglia, SuperMon, Astrolabe, Borealis, and MRNet, use data aggregation to synthesize large data sets and reduce data volumes while retaining relevant information content. Hierarchical or tree-based overlay networks (TBONs) are often used to execute data aggregation operations in a scalable, piecewise fashion. In this paper, we present *state compensation*, a scalable failure recovery model for high-bandwidth, low-latency TBON computations. By leveraging inherently redundant state information found in many TBON computations, state compensation avoids explicit state replication (for example, process checkpoints and message logging) and incurs no overhead in the absence of failures. Further, when failures do occur, state compensation uses a weak data consistency model and localized protocols that allow processes to recover from failures independently and responsively.**

Based on a formal specification of our data aggregation model, we have validated state compensation and identified its assumptions and limitations: state compensation requires that data aggregation operations be associative, commutative and idempotent. In this paper, we describe the fundamental state compensation concepts and a prototype implementation integrated into the MRNet TBON infrastructure. Our experiments with this framework suggest that for TBONs supporting up to millions of application processes, state compensation can yield millisecond failure recovery latencies and inconsequential application perturbation.

***Keywords*-component; formatting; style; styling;**

I. Introduction

Many high-performance computing (HPC) tools and applications use data aggregation to synthesize massive data sets from many distributed sources. In these contexts, data aggregation is used to reduce data volumes without losing relevant information as well as to reduce data analysis requirements. Hierarchical or tree-based overlay networks (TBŌNs) that can aggregate data in a piecewise fashion can easily achieve large-scale parallelism making them popular for high-bandwidth, low-latency data aggregation. Parallel prefix computations [16] are an early example, and today TBŌN-based aggregation can be found in general aggregation services [5], [7], [11], [24], distributed debugging, performance and monitoring tools [20], [25], [26], [28]; information management systems [22], [29]; stream processing [6]; and mobile ad hoc networks (MANETs) [17], [30]. The aggregation operations range from *sum*, *average*, and *upper/lower bound* computations to complex ones like clock synchronization, data classification, and data unification (e.g. subgraph merging).

Many applications of data aggregation require high degrees of robustness. For example, data analysis based on anomaly detection, as in tools for finding outliers in large scale computing simulations[1] or detecting anomalous traffic patterns in large data networks, might fail to capture outlier behavior if only partial data is aggregated. Additionally, the TBŌN data aggregation model is generalized in stream programming environments such as Borealis[6] and SDIMS [29], which have shown how this computational model can be used for applications like data-diffusion, publish-subscribe systems, barrier synchronization and voting. Once again, these applications often require total data aggregation. However, expected failure rates of existing and imminent petascale systems have raised serious concerns over current recovery models: at these scales, checkpoint-based mechanisms will consume most of the available computational resources allowing little or no application work to be done [10], [12]. The scalability of checkpoint-based fault tolerance is limited for two main reasons: (1) the time necessary to checkpoint local process state increases linearly with process size (which generally increases with memory capacity) and (2) the worse case coordination time to create a globally consistent set of checkpoints increases quadratically with the number of processes.

In response to these concerns, we have developed a technique called *state compensation* for robust, high-performance data aggregation in the face of transient or permanent *fail-stop* failures, detectable failures that cause processes to cease output production. Our cen-

[1] In III-B we describe one such tool, STAT.

tral observation is that many TBŌN-based computations naturally maintain redundant state amongst the processes in the system. Intuitively, 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. State compensation uses the redundant state from processes that survive failures to compensate for lost information, thereby avoiding explicit data replication (like checkpointing or message logs). State compensation's general requirements are that operations be associative and commutative.

In this paper, we focus primarily on one compensation mechanism, *state composition*. In state composition, orphaned processes propagate their local aggregation state, to their new parents to compensate for any data lost in transit from the orphans to their failed parent or from the failed parent to the failed parent's parent. State composition is appropriate for *idempotent* operations for which re-processing some input elements does not change the computation's output. We also summarize the results from a preliminary study of a second mechanism, *state decomposition*, that accommodates non-idempotent operations by precisely computing and compensating only for the lost state. Both mechanisms use a weak data consistency model and localized algorithms (for example, tree reconfiguration) that allow processes to recover from failures independently and responsively. Overall, this paper describes the state compensation model, a working implementation of state composition added to the publicly available MRNet TBŌN prototype [24], and an evaluation based on this prototype. This evaluation shows that for TBŌNs supporting up to millions of application processes, state composition can yield millisecond failure recovery latencies with unnoticeable application perturbation.

After related work in Section II, we present the TBŌN computational model and its key properties in Section III. In Section IV, we describe the state compensation mechanisms followed by our MRNet-based prototype in Section V. After our empirical performance evaluation in Section VI, we summarize our contributions and discuss open issues.

II. Related Work

Related recovery models can be categorized as *fail-over*, *rollback recovery*, or *reliable data aggregation*. In fail-over (or hot backup) protocols [1], [27], processes periodically synchronize with backup replicas used to replace failed primaries. This approach can be applied generally to any computation and yields low recovery latencies since backups are in (near) ready states. However, with only one backup per primary, the resource overhead is 100%. Additionally, primary/backup synchronization increases normal operational latencies.

In rollback recovery, processes periodically checkpoint their state to persistent storage. Upon failures, the system recovers to the most recent checkpoint [9]. In coordinated checkpointing, the common distributed rollback recovery variant, processes coordinate to record globally consistent checkpoints. Like hot backups, rollback recovery is a well studied, general fault tolerance mechanism. However, as previously discussed, recent studies conclude that for imminent petascale systems, checkpointing and recovery resource demands will prevent applications from performing useful work [10], [12].

Various recent studies specifically have targeted reliable data aggregation in domains including stream processing engines (SPEs), distributed information management systems (DIMS) and mobile ad hoc networks (MANETs). In SPEs, fail-over protocols have been used to replicate query processing nodes [6], [14]. In Astrolabe [22] and Gupta et al.'s approach [13], nodes are organized into disjoint clusters, and a hierarchy is imposed by forming larger clusters of clusters. Data are replicated within and across the clusters using periodic, random *gossiping* [23]. SDIMS [29] also organizes nodes into a tree but uses an explicit replication protocol in which nodes scatter attribute information to their ascendants and descendants. These approaches are general and can be used in any data aggregation context. Additionally, gossip protocols are scalable with a configurable trade-off between overhead and robustness. However, gossip protocols are best suited for applications with small data sets (hundreds of bytes) that do not require low latency communication and can tolerate partial, non-deterministic output. Further, the explicit replication of the SDIMS approach has potentially high space overhead as data are replicated in multiple places.

Robust MANETs [15], [18], [19], [21], use unreliable transport protocols periodically to disseminate locally known attribute data. Nodes merge received attribute data with their local data such that over time local attribute estimates converge to their actual values. These protocols generally exhibit good scalability and convergence rates. However, proposed protocols are specific to relatively simple aggregation operations like *sum*, *max* and *average*.

Discussion: Other than MANET approach, these related works can be applied generally to data aggregation operations. However, fail-over and rollback recovery protocols are inherently non-scalable. In contrast, the gossiping approaches are scalable, but not suited for high-performance operation and yield partial, non-deterministic output. Our recovery model requires as-

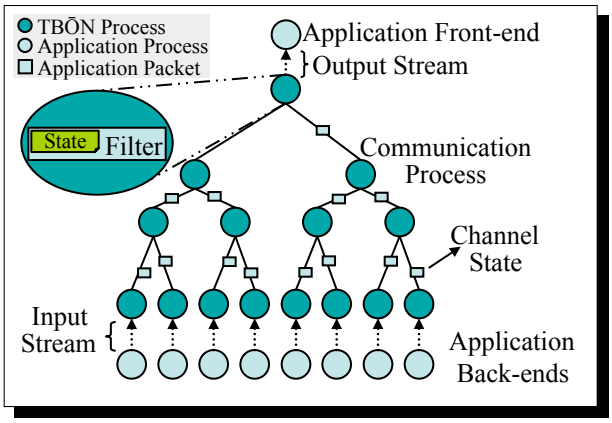

Figure 1. TBŌN Model: back-ends stream data via communication processes that aggregate and propagate the data to the front-end.

sociative, commutative (and idempotent) aggregation operations. However, by leveraging information redundancies inherent to the computation, state compensation avoids explicit state replication bearing no additional overhead during normal operation. We are unaware of any other recovery model that does not rely on explicit replication. Additionally, state compensation features deterministic, high throughput delivery without constraints on message sizes. Finally, recovery is fast, involves a small subset of the TBŌN and yields inconsequential application perturbation.

III. The TBŌN Model

Generally, TBŌNs aggregate data in a piecewise fashion by iteratively sub-dividing the input data. As shown in Figure 1, *application*[2] *back-end processes* propagate continuous dataflows via the TBŌN to the *application front-end*. Hierarchically organized *communication processes* implement the TBŌN's communication and aggregation services. MRNet [24], our TBŌN prototype, has two main components: *libmrnet*, a C++ library linked into the application processes and *mrnet_commnode*, the program for the communication processes. While balanced trees are typical, MRNet supports any connected tree topology. Processes exchange data via a reliable, order-preserving transmission; for example, MRNet uses TCP. Applications use *streams* to distinguish logical dataflows. Streams specify the participating back-ends and the aggregation to apply to the stream's packets. MRNet supports simultaneous streams each with potentially different filters.

A. Data Aggregation

TBŌN processes use *filters* to aggregate data from their children. Our 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 over time state size can become large. Generally, this filter state encapsulates previously filtered inputs and is used to propagate incremental updates efficiently. For example, consider the *sub-graph folding* filter [25], 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 data arrive, the filter outputs changes to its current merged graph. MRNet filters input a packet vector and output a packet vector (typically of a single packet):

```
void filter( vector<Packet> &inPackets,
             vector<Packet> &outPackets,
             void **inoutFilterState );
```

The filter also takes a filter state reference, *inoutFilterState*. MRNet maintains a unique reference for each filter instance. Filters can allocate memory for persistent data and point the reference to this data. MRNet passes the reference to the filter on each invocation.

MRNet's *load_FilterFunction()* routine dynamically loads new filters into the TBŌN processes from shared object files. MRNet supports various *synchronization modes* to deal with asynchronous input packet arrival. In this paper, we assume the common *Wait For All* mode in which an *input wave* is comprised of a single packet from every child process.

State join and difference: We further abstract the filter function into join and difference operations. The join operator, $\sqcup$, is used to merge input packets into waves and to update the filter state with these input waves. We assume that the join operator is associative and commutative. The relevance of associativity has been noted previously, for example, in parallel prefix computations [16]. As detailed in Section IV-A, these properties simplify our recovery mechanisms: since correctness does not depend on the grouping and ordering of input data, disconnected subtrees may reconnect to any branch of the main tree. We assert that these properties of associativity and commutativity are not overly restrictive: in practice, we have not observed a single filter for which the correctness of its output data that was dependent upon the ordering or grouping of the filter's input data.

State composition assumes that joins are idempotent, $\forall x, x \sqcup x = x$. Many stateful aggregation operations, including most MRNet-based data aggregation operations [3], [4], [24], [25], are idempotent. Specific algorithms classes include set union, graph folding, equivalence class computations, and upper and lower bounds computations. Variations of these operations that include membership statistics, for example set

[2]"Application" refers to the system directly using the TBŌN, whether it is a tool or application.

union with membership counts, are non-idempotent. (State decomposition addresses non-idempotence (Section IV-D).)

Filters use the difference operator, $-$, to compute the incremental difference between their previous and updated states. Filters based on idempotent joins may output either incremental or complete updates. For efficiency, we favor the latter. However, as we see in Section IV-C, sending complete updates simplifies the state composition protocol. The inability to do this in non-idempotent operations encumbers state decomposition.

B. An MRNet-based TBŌN Example

As an example, we describe the MRNet-based stack trace analysis tool (STAT [3]), which aggregates callgraphs sampled over time from large MPI applications to identify equivalence classes of process behavior. STAT back-ends propagate callgraphs through the TBŌN to the STAT front-end. Each MRNet process uses a custom STAT filter to merge new stack traces with those previously filtered and propagate the incremental callgraph changes. The persistent state at each process is a callgraph prefix tree representing the history of filtered callgraphs. The state of each channel is the incident vector of pending updates from a child to its parent. The final output at the application front-end is a callgraph prefix tree representing a global profile of the application process' behavior. This example is representative of many other types of data aggregation operations, for example, sub-graph folding [25] in the Paradyn tool and data equivalence classification [24].

IV. State Compensation

State compensation works by merging states from non-failed processes to compensate for lost state. Before describing the details, we discuss our failure and consistency models and the fundamental TBŌN properties upon which state compensation depends.

A. Failure and Data Consistency Models

State compensation tolerates fail-stop failures in any root, internal or leaf TBŌN process. These processes may fail at any time, even simultaneously. Failed processes need not be replaced: the TBŌN is reconfigured to omit failed processes, but new processes may be integrated dynamically. Should all TBŌN processes fail, the system degenerates to a flat tree with the front-end directly connected to the back-ends. Network failures that cause permanent partitions are treated as failures of the processes partitioned from the front-end.

State compensation does not address failures in application processes; such processes may be viewed as sequential data sources and sinks amenable to sequential checkpointing, which avoids the non-scalable coordination complexities of distributed checkpointing. This solution does not work if the application uses non-TBŌN communication channels, but we have not yet seen this case in practice. We might expect failures to be more common in back-end processes, which outnumber communication processes. However, a planned MRNet extension will support a process separation between the MRNet functionality now implemented in the back-end library and the back-end process. This will allow a dedicated process to responsively attend to high speed data transmissions without adding MRNet threads that may erroneously interfere with already multithreaded back-ends. The collocated communication process and back-end will communicate efficiently via shared memory. This extension will result in a one-to-one mapping of leaf communication processes to back-ends rendering TBŌN processes as failure prone as back-ends.

Normal:	7	11	27	35	35
Failure:	7	8	15	35	35
	t_0	t_1	t_2	t_3	Maximum

Figure 2. Convergent Recovery: A failure at t_0 causes a divergence at t_1 and t_2. The output re-converges at t_3.

State compensation guarantees weak data consistency, called *convergent output recovery* [14], 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. Figure 2 shows the output streams of two originally identical TBŌNs executing an *integer maximum* aggregation: the front-end continuously outputs new maxima 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.

B. Fundamental TBŌN Properties

To guarantee the correctness of our recovery mechanisms, state compensation relies on three TBŌN properties: (A full theoretical discussion is attached as an appendix.)

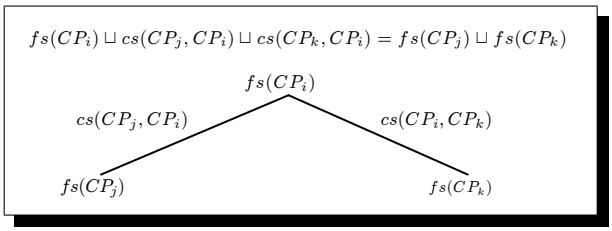

Figure 3. Inherent Redundancy: a parent's filter and pending channel states equals its children's states.

Inherent Redundancy: Inherent information redundancies exist in stateful TBŌN-based data aggregations. Intuitively, as data is propagated from the leaves toward the root, filter state is replicated at successive levels in the tree. Formally, our *Inherent Redundancy Lemma* states: "the join of a parent's filter and pending channel states equals the join of its children's states". This is shown in Figure 3, where CP_i represents the i^{th} communication process, $fs(CP_i)$ represents CP_i's filter state, and $cs(CP_i, CP_j)$ represents the pending updates from CP_i to CP_j. Once a parent, in this case CP_i, has filtered all the updates from its children, the parent's filter state will be equal to the result of joining its children's filters state – they would have filtered precisely the same history of information.

All-encompassing Leaf States: The All-encompassing Leaf States Lemma 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". 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 necessarily subsume the state and updates throughout the rest of the TBŌN.

TBŌN Output Dependence: Finally, the *TBŌN Output Dependence Lemma* states: "the output of a TBŌN computation is solely a function of the root's state and the TBŌN channel states." Intuitively, in-flight data triggers the execution of data aggregations, the output of which depends upon this data and the current state of the filtering process.

C. 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, 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. Formally, our *State Composition Theorem* states: "A TBŌN can tolerate failures without changing the computation's semantics by re-introducing filter state from the orphaned processes as channel state." A sketch of the proof (attached in the appendix) follows: Again considering the TBŌN in Figure 4, from the *TBŌN Output Dependence Lemma*, we know that the output of CP_i, the root, only depends upon its state and the subtree's channel states. Since the *All-encompassing Leaf States Lemma* tells us that all the states in this subtree are subsumed by the states at the leaves, CP_m and CP_n, then re-introducing these states (after reconfiguration) as channel state compensates for all the lost state. Since CP_m and CP_n's states may contain information already filtered by CP_i, the aggregation operation must be idempotent, that is, resilient to re-filtering some input data elements.

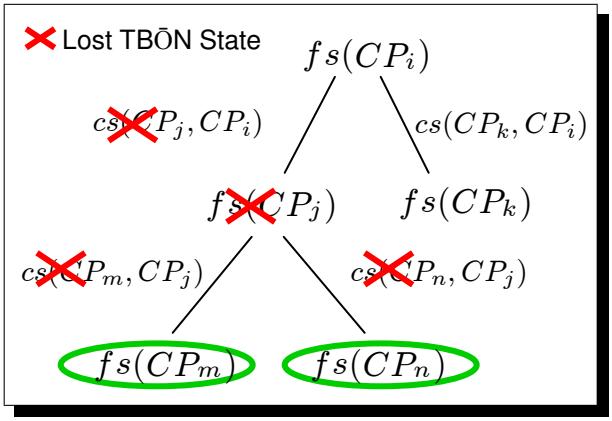

Figure 4. State Composition: When CP_j fails, $fs(CP_j)$, $cs(CP_j, CP_i)$, $cs(CP_m, CP_j)$, and $cs(CP_n, CP_j)$ are lost. $fs(CP_m)$ and $fs(CP_n)$ compensate for the loss.

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 items that constitute 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 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.

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.

D. State Decomposition

We also have done a preliminary study of a second mechanism, *state decomposition*, 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 of the failed process' parent and the failed process' siblings' and children, decomposition precisely computes what input information from the orphans had been filtered by the parent of the failed process. In other words, after accounting for data the failed process' parent has filtered from its non-failed children, the difference between the orphans' filter states and the state at the failed process' parent reflects the lost state. After reconfiguration, decomposition can precisely compensate for this lost state. A more detailed discussion of decomposition is beyond the scope of this paper.

V. New MRNet Fault-Tolerance Extensions

The major components of state composition are failure detection, tree reconfiguration, and lost state recovery. Accordingly, the major MRNet extensions are an event detection service for failures and other events, a protocol for dynamic topology (re-)configuration and an implementation of the state composition compensation mechanism.

A. The MRNet Event Detection Service

We implemented a passive, connection-based mechanism to detect important asynchronous system events like process failures or adoption requests. This avoids the overhead associated with active probes. The new MRNet event detection service (EDS) runs as a thread within each process and primarily monitors a *watch list* of designated *event sockets*. The EDS passively monitors these sockets using the *select* system call to wait until a specified event occurs on at least one monitored socket. These sockets include a *listening socket* to which other TBŌN processes can connect. The two primary protocol messages delivered to an EDS are the *New Failure Detection Connection* protocol message, used to establish event sockets for failure detection, and the *New Data Connection* protocol message, used by orphans to request adoptions.

Failure Detection: For component failure detection, centralized approaches and approaches that require coordination amongst many processes do not scale. Therefore, we leverage the TBŌN structure to establish small groups of processes that monitor each other. Each MRNet process monitors its parent and children. Therefore, the number of peers each process monitors is determined by its fan-out. A newly adopted process sends its parent the *New Failure Detection Connection* protocol message. Both the child and parent add the sockets used to send and receive this message to their watch lists as *failure detection event sockets*. When a process fails, its host's kernel abort its connections, and these connection abortions are detected immediately by the process' remote peers. User-level heartbeat protocols [2] could be used for responsive, user-controlled host and link failure detection.

Dynamic Topology Configuration: We needed to extend MRNet with support for dynamic topology configurations to accommodate tree reconfigurations after process failures. While the complete details of our tree reconfiguration study will be the topic of

a future paper, we describe the highlights. We desired an algorithm that could execute quickly, have manageable data requirements, execute at orphans without coordination and still yield well-performing tree configurations. Not surprisingly, our analysis of TBŌN data aggregation performance suggested that balanced, shallow topologies yield the best performance. After comparing several strategies of varying complexities, we chose one that required each orphan to maintain the entire topology. At startup, each TBŌN process receives the entire topology, a set of nodes and edges. A tree with a fan-out of 32 and a height of 4 supports 1,048,576 back-ends and requires 32^3 communication processes. Representing a communication process requires 10 bytes of data (4-byte rank, 4-byte IP address, 2-byte port) and representing edges requires 8 bytes (2 4-byte ranks). The broadcast of this 600 kilobytes of data is a manageable one time cost at TBŌN startup, and more compact graph representations could further decrease communication costs. Topology updates, described below, are small and expected to be infrequent.

When a failure occurs, an orphan generates a list of *potential adopters* such that if any member of that list were to adopt that orphan, the depth of the tree would not increase. That is, the depth of each potential adopter's subtree must be equal or greater than that of the orphan's failed parent. Each potential adopter is sorted by a weight based on its fan-out and subtree height. Then to mitigate the pathological cases where most orphans prefer the same adopter, orphans independently use their rank to effect a round-robin adopter selection. Preferred adopters still adopt the most orphans, but this guarantees that the maximum difference between the number of adoptions by any two adopters is one. The runtime latency of this algorithm for our TBŌN that supports 32^4 back-ends is on the order of hundreds of milliseconds. Orphans connect to its chosen adopter's EDS and send the *New Data Connection* protocol; the adopter and adoptee establish a socket for application data transmission.

Topology Updates: Topology modifications due to failures and adoptions must be broadcast to all TBŌN processes. Just as for application data, we use the tree structure for efficient, scalable dissemination. Propagation of failure reports with the failed process' rank is initiated by detecting EDSes. Adoption reports with the adopter's and adoptee's ranks are initiated by the adopting and adopted processes. Upon receiving a report, a process propagates it to all its peers other than the one from which it received the report.

Reconfigurations and propagation delays can lead to duplicate, late, missing or out-of-order topology update reports. For example, a process adopted multiple times can receive the same report multiple times from different branches. Applying duplicate failure or adoption reports result in the same updated topology.

Untimely failure reports lead to stale topology information. Our reconfiguration algorithms tolerate this by iteratively connecting to potential adopters, which may have failed, until an adoption succeeds. A missing failure report is infinitely late. Untimely adoption reports can lead to erroneous cycles if an orphan is adopted by a process in the orphan's subtree. Our current prototype does not include the transaction mechanisms necessary to avoid such cycle formation, though orphans perform a topology validation that can avoid some instances. Late adoption reports may also lead to functionally correct but sub-optimal topologies, for example, if a process using a stale topology computes that an adoption would not lead to an increased tree height when, in fact, it would.

Different failure reports cannot conflict, so out-of-order failure reports are acceptable. However, reconfiguration reports of different adoptions regarding the same orphan are conflicting. If processed in the wrong order, topology information will become incorrect. Each process maintains an *incarnation number* [8] incremented every time the process is adopted and propagated with the adoption report. Processes disregard adoption reports of orphans for which they have received a report with a higher incarnation number.

B. State Composition Implementation

Our state composition theory leads to a straightforward implementation in which orphaned processes are the primary actors. When a child detects its parent's failure, the orphaned child must re-establish a path to the application front-end and compensate for any lost state. In the current implementation, an orphan pauses input data processing until it is adopted by a new parent. Alternatively, an orphan could continue to fetch and filter new input and buffer its output until it has been adopted. In fact, since aggregation is assumed commutative and associative, upon adoption it would be sufficient for an orphan to propagate the aggregate of its pending output instead of individual output packets. After input data processing is paused, the orphan initiates a TBŌN reconfiguration by iteratively probing for non-failed adopters, as previously described. After the reconfiguration, the adoptee compensates for any lost state by propagating its filter states to its adopter.

A parent process that detects a child's failure simply deletes the failed process from its topology data structure. After the recovery is completed, failure and adoption reports are disseminated.

1) MRNet Interface Extensions: We added to MRNet a *get_FilterState* routine to distinguish filter functions that comply with state composition and to extract the compensating filter state during recovery:

```
outPacket get_FilterState(void **inFilterState);
```

The *get_FilterState* routine, also implemented by the filter implementor, is passed an *inFilterState* pointer that references the filter state that MRNet manages for each filter instance and returns a packet that contains the filter state's data. During state composition, the returned *outPacket* is sent to an adopted process' new parent. We augmented the new *load_FilterFunction* routine to query its input shared object for a *get_FilterState* routine for the loaded filter function. If a *get_FilterState* routine is found, the filter function is designated as *state recoverable*. MRNet does not attempt to compensate for lost state in operations not designated as state recoverable.

VI. Evaluation

In the absence of failures, state compensation consumes no computational resources beyond those of normal operation. Therefore, we only evaluated failure recovery performance 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 8-CPU AMD Opteron nodes linked by a double data rate InfiniBand network.

A. The Experimental Framework

We implemented 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 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.

B. The Application

We test failure recovery with our integer union computation introduced in Section IV-A, which computes the set of unique integers in the TBŌN's input stream by filtering out duplicates. This computation has easily verifiable output, and executes an equivalence classification similar to many useful, more complex aggregations. The application back-ends propagate randomly generated integers at a ten hertz, the default sampling rate of the STAT tool [3]. After each experiment, we compare the input from the back-ends with the output at the front-end; the output set must be equal to the union of the input sets.

C. 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, and $l(cleanup)$ is the time to update local data structures and propagate topology updates.

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.

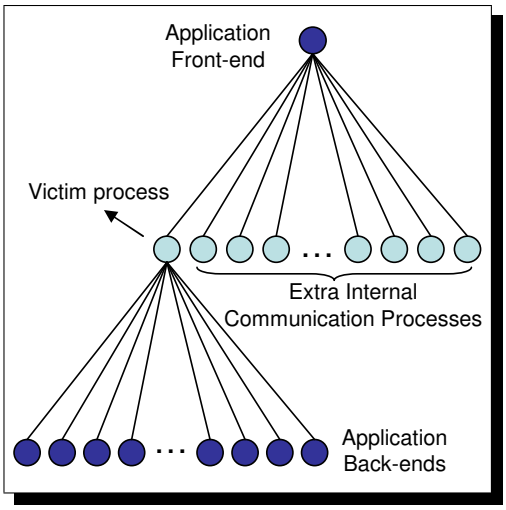

Figure 5. Micro-benchmark Topology: The victim has the fan-out being evaluated, and 16 internal processes are added to distribute orphan adoptions.

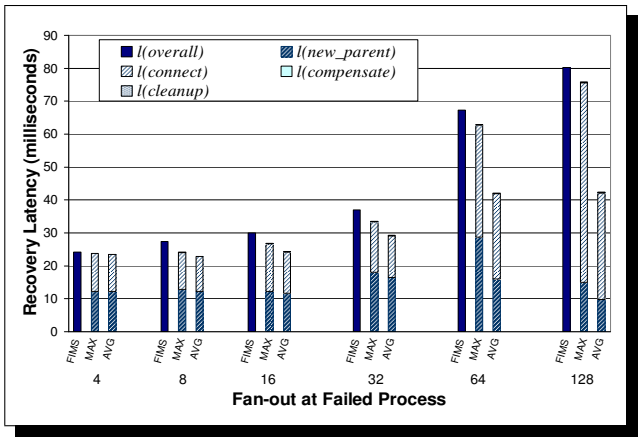

Figure 6. Recovery Latencies: $l(overall)$ is overall TBŌN recovery estimate, $l(new\_parent)$, $l(connect)$, $l(compensate)$ and $l(cleanup)$ are the times to choose parent, connect, send filter state, and do cleanup.

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 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 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 – insignificant considering that a 128^3 tree has over 2 million leaves.

D. 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. For increased data rates, with millisecond recovery latencies we still expect little performance perturbation.

VII. Conclusion

State compensation is 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. Our state composition mechanism requires only that data aggregations are commutative, associative and idempotent and is suitable for many useful algorithms and promises to scale to TBŌNs that can support millions of processes. As HPC system sizes and failure rates continue to increase, no (and low) overhead recovery models like state compensation are essential. We plan to extend this work by studying compositions of heterogeneous filter functions, and completing the implementation and empirical evalu-

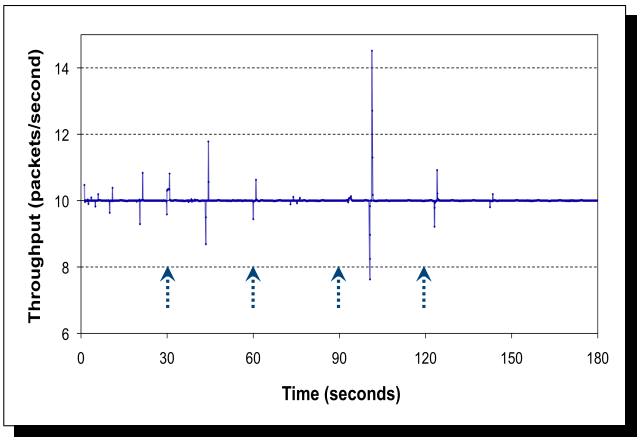

Figure 7. Application Perturbation: Failures (indicated by arrows) are injected every 30 seconds into a TBŌN with an initial 32^3 topology.

ation of state decomposition and fault-tolerance for application back-end failures. Also, many of our current aggregations are motivated by the analysis requirements of parallel and distributed system tools. Open questions are how will 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.

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.

[2] G. R. Andrews. Paradigms for process interaction in distributed programs. *ACM Computing Surveys*, 23:49–90, 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] B. Badrinath and P. Sudame. Gathercast: the design and implementation of a programmable aggregation mechanism for the internet. In *9th International Conference on Computer Communications and Networks*, pages 206–213, Las Vegas, NV, October 2000.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] D. A. Evensky, A. C. Gentile, L. J. Camp, and R. C. Armstrong. Lilith: Scalable execution of user code for distributed computing. In *6th IEEE International Symposium on High Performance Distributed Computing (HPDC 97)*, pages 306–314, Portland, OR, August 1997.

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

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

[14] 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.

[15] 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.

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

[17] 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)*, 2002.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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)*, March 2006.

[26] 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*, pages 289–298, Hong Kong, September 2003.

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

[28] M. J. Sottile and R. G. Minnich. Supermon: A high-speed cluster monitoring system. In *IEEE International Conference on Cluster Computing (CLUSTER 2002)*, pages 39–46, Chicago, IL, September 2002.

[29] 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.

[30] Y. Yao and J. E. Gehrke. Query processing in sensor networks. In *First Biennial Conference on Innovative Data Systems Research (CIDR 2003)*, Asilomar, CA, January 2003.