LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

UCRL-CONF-222793

State Compensation: A No-cost Scheme for Scalable Failure Recovery in Tree-based Overlay Networks

D. C. Arnold, B. P. Miller

July 12, 2006

Second Workshop on Hot Topics in System Dependability (HotDep '06)
Seattle, WA, United States
November 8, 2006 through November 8, 2006

State Compensation: A No-cost Scheme for Scalable Failure Recovery in Tree-based Overlay Networks

Dorian C. Arnold and Barton P. Miller
Computer Sciences Department, University of Wisconsin
Email: {darnold, bart}@cs.wisc.edu

Abstract

Tree-based overlay networks (TBŌNs) have become important for scalable data multicast and aggregation. This infrastructure's generality has lead to widespread usage in large scale and widely distributed environments – environments in which reliability must be addressed. This paper presents *state compensation*, a novel reliability concept for TBŌN environments that avoids explicit state replication (such as checkpoints) for failure recovery by leveraging general properties of TBŌN computations that allow computational state from non-failed processes to compensate for state lost from failed ones.

In this paper, we present our state compensation mechanisms, prove sufficient properties of distributed computations that make these mechanisms feasible and show how to derive computation-specific recovery primitives from these properties. We also present a case study of the recovery process. The result is a general TBŌN recovery model that requires no additional storage, network, or computational resources during normal operation.

1 Introduction

As high-performance computing (HPC) trends toward petascale power, supercomputers and computational Grids with 10^5 and even 10^6 processors are becoming more common: topped by LLNL's BlueGene/L with 131,072 processors, the systems on the June 2006 Top500 averaged 1747 processors and 31.4% (157 systems) had at least 1024 processors [1]. As HPC systems continue to get larger, application and tool developers face two major challenges: scalable performance and system reliability.

TBŌNs[1] for performance Tools and applications that perform well at low scales eventually will experience computational and I/O bottlenecks as they run at increasingly larger scales. These bottlenecks lead to the system's scalability ceiling – the point beyond which additional computational power will not improve significantly the system's performance. Such scalability concerns have lead to the emergence of tree-based overlay networks (TBŌNs) as an important computational model for scalable distributed computing. A TBŌN is a hierarchal organization of processes used to implement data multicast, gather and aggregation services. For data aggregation, *filters* are used throughout the TBŌN; filters may use *persistent state* to relay information to subsequent invocations. The flexibility of filters make them useful for obvious aggregations, like *min*, *max*, *count*, and *average*, as well as being surprisingly useful for many more complex ones.

Tree overlays (called multicast trees) had originally been used for scalable multicast infrastructures [10]. Now TBŌNs are becoming popular as middleware for data gathering and aggregation services [3, 4, 7, 9, 11]. In contrast to these general-purpose infrastructures, several distributed tools use TBŌNs in application specific modes. For example, Ganglia [13] and Supermon [14] use a multi-level process hierarchies for efficient collection and aggregation of system monitoring data.

More recently, we have shown that many algorithms from areas like image processing, information retrieval, and bio-informatics can use the TBŌN model for scalable performance [2]. Two key result of that work are: 1) many commonly used data clustering and anomaly detection algorithms are based on an *equivalence* computation in which some abstraction of equivalence is used to determine elements' similarities, and 2) equivalence computations are a good fit for the TBŌN model – critical facts as we will show that such computations meet the requirements of our reliability mechanisms.

The reliability problem As system size increases, the mean time between failures decreases; therefore, reliability must be addressed for effective use of large scale systems. However, to limit application perturbation, we need mechanisms with low-overhead and resource utilization.

Concerns have been raised about the scalability of traditional fault-tolerant techniques that rely on checkpoints [6] or fail-over protocols [5]. For example, based on a recent study, Elnozahy and Plank conclude that for systems projected to be available by 2010, checkpoint protocols will require dedicated resources and that poorly

[1] Although this term is not prevalent in the literature, we feel it offers an intuitive and accurate description of the computational model.

chosen checkpoint intervals may lead to overloaded network and storage resources.

Given the increasing popularity of TBŌN infrastructures and the need to address reliability for petascale systems and beyond, we aim to develop scalable reliability mechanisms for data aggregation in TBŌN environments, targeting non-transient, fail-stop process/node failures. While there has been much research on reliable multicast, we are unaware of any other work specifically focused on reliable aggregation.

Scalable TBŌN Reliability Gärtner observes that a system cannot be reliable without spatial or temporal redundancy [8]. For example, checkpointing employs both – spatial redundancy to record program state and temporal redundancy to repeat a portion of the system's execution after a rollback occurs. Our position in this paper is that for important classes of TBŌN computations, we can get spatial redundancy for free and use efficient operations to roll forward a failed process' state (potentially beyond its state at the time of failure by incorporating state that the failed process would have seen eventually). The result is a scalable reliability model for TBŌN environments.

Our approach to TBŌN reliability, *state compensation*, is based on two observations: 1) there exists an inherent information redundancy amongst the computational states at TBŌN processes; and 2) for certain computations, state lost due to failure may be replaced by non-identical state with little or no effect on the computation – a concept we call *computational consistency*. We combine these observations into the following strategy: compensate for lost process state using redundant information from non-failed processes; recovered state need not be identical, only computationally consistent, to the state it replaces.

To take advantage of this strategy, we require that the persistent state stored in each communication process and the input/output packets of the TBŌN computation have the same representation either directly or through known mappings. Also, the filter computation must be distributive and associative. These requirements are met by all stateful filters we have seen in practice, most notably, all the equivalence computations that we have evaluated. The result is a set of failure recovery mechanisms that do not require **any** additional storage, network, or computational resources during normal execution. Other features of our reliability strategy are relaxed consistency models based on computational consistency, and failure confinement where only a small subset of the entire tree participates in recovery. Lastly, the mechanisms are broadly applicable: having previously shown that many useful algorithms reduce to *equivalence computation* [2], we show that such computations are well-suited for our methods.

In the rest of this paper, we define our TBŌN environment and its relevant fundamental properties in Section 2. In Section 3, we make the case for state compensation by presenting the definitions, proofs and requirements as well as a demonstration of how the filter function itself can often be used for this operation. We conclude with a case study and a discussion of practical issues.

2 TBŌN Model Fundamentals

In this section, we describe the TBŌN model, relevant notation, and two fundamental TBŌN concepts for our recovery methods: natural information redundancies and computational consistency.

In the TBŌN model, depicted in Figure 1, application processes (processes from the system directly leveraging the TBŌN) are connected by a tree of N internal or *communication processes*, $\{CP_0, CP_1, \ldots, CP_N\}$. Collectively, the root and leaf processes are called *end-points*. This tree of processes, connected via FIFO channels, serve as conduits through which application-level packets flow amongst end-points. Upstream flows propagate toward the root, downstream flows away.

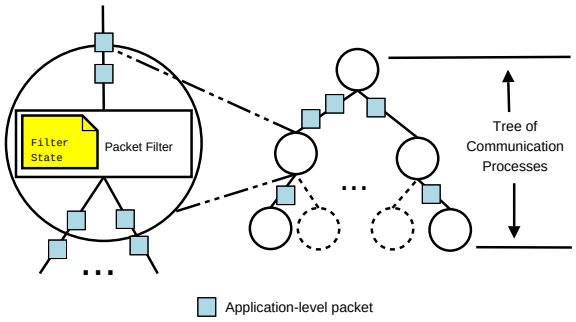

Figure 1: The components of a TBŌN.

Data filters permit the placement and control of application-level logic throughout the TBŌN infrastructure. This feature is used to perform data aggregation operations on packet flows. A filter, f, can be any function that inputs a set of packets and outputs a single (potentially null-valued) packet[2]. Persistent filter state, used to carry information from one filter execution to the next, increases the power of the filter abstraction. $fs_p(CP_i)$ is CP_i's state after p filter invocations.

Naturally, as packets flow upstream, output from children become input to parents. The next pending input from child CP_j to parent CP_i is labeled $in(CP_i, CP_j)$. For convenience, we use $IN(CP_i)$ to designate the set of next pending packets from all children of CP_i. Analogous to input, $out(CP_i)$ is the filter output.

Using our notation, a filter, f is defined as:

$$\{out(CP_i),\ fs_{p+1}(CP_i)\} \equiv f[IN(CP_i),\ fs_p(CP_i)] \quad (1)$$

[2]The general TBŌN model does not preclude multiple filter outputs, but in practice we have not seen the need for it.

That is, a filter inputs the next packet from every child and the current filter state and produces a new output while updating its state.

As discussed in Section 1, the TBON model has been used by many systems, all of which use standard data filter operations like *min*, *max*, *average*, and *count*, and several allow user-defined filters. Previously, we have demonstrated more complex filter computations for distributed clock-skew detection, time-aligned data aggregation, subgraph folding for scalable data presentation, and creating data histograms [11, 12], and more recently for data clustering/partitioning [2].

For stateful filters, persistent state generally captures the aggregated effect of processed inputs; new inputs/outputs become incremental updates to this state. This is shown with a simple yet representative example, the *unique integer* filter. This filter outputs the list of unique values in its input stream using integer equivalence. The state at each process is the history of unique values. As inputs arrive, the filter may output the newly calculated list of unique values or just the latest updates. This example is particularly relevant since given large classes of TBON computations reduce to integer equivalence [2], **if we solve the reliability problem for integer equivalence, we do so for many computations!**

Inherent Information Redundancies There exists a natural information redundancy amongst the persistent states of TBON processes in aggregation networks. This redundancy occurs generally in TBON computations because the persistent state at each communication process, a summary of the input stream filtered by that node, is naturally replicated by communication processes as they successively filter the input data as they propagate upstream.

Computational Consistency While a complete discussion of computational consistency is beyond the scope of this paper, we offer the intuition behind it. Essentially, two computations are computationally consistent if given the same initial state and input stream, they are guaranteed to yield (eventually) identical output even if they transition through different intermediate states. Regarding our recovery model, our protocols ensure that as the TBON continues to process its input stream after a failure recovery, the output stream eventually will return to match what it would be had the failure not occured, though some intermediate (typically expendable[3]) results may be omitted.

3 State Compensation

For computations that fit our model of computational consistency, we can define recovery operations based on *state merging* (merging computational states from TBON processes) to compensate for state lost due to failure. In this section, we outline our recovery model and define *state composition*, the focal compensation operation of this paper. We then prove the key properties of TBON computations that lead to state composition and derive composition operations based on these properties.

Our strategy to recover from failure is: 1) detect a failure, 2) reconstruct the process tree to accommodate the failure, 3) generate compensatory state for any lost state, 4) reintegrate the regenerated state into the tree and, 5) resume normal operation. Failure detection and tree reconstruction issues are orthogonal to our recovery model and not discussed in this paper. We now describe Steps 3 and 4, the use of filter state operations to regenerate and re-integrate lost state into the TBON.

State Composition State composition is motivated by situations where we want to reconstruct the state of a failed parent from those of its children. The idea is similar to the piecewise deterministic concept which states that a computation can be pieced into sequences of deterministic execution each starting with a non-deterministic event, typically a message receipt, and that these messages can be logged for exact replay of the computation upon failure [15]. In state composition, the (effects of a) process' input messages are captured in the filter states of its children which are used to replace the parent's state after a failure. Using a binary tree (without loss of generality) and $\oplus$ as the composition operator, composition becomes:

$$fs_p(CP_i) \equiv fs_q(CP_j) \oplus fs_r(CP_k). \qquad (2)$$

That is, given a TBON in which CP_i is the parent of CP_j and CP_k, the composition of CP_j and CP_k's states is a compensatory state for CP_i. By the nature of a TBON, downstream processes will have filtered more input than upstream processes. Additionally, different processes at the same level in the tree may execute at different rates. So for Equation 2, $q, r \geq p$ and q does not have to be equal to r. When filter states from children are composed, the generated state represents that of a parent *synchronized* with its children. Informally, two processes are synchronized if there are no pending messages between them. In other words, the regenerated filter state at the parent reflects the parent having filtered all the input data that its children have filtered. **This means that state composition also compensates for the messages lost due to failure!**

This approach, however, works only when the filter computation is *idempotent*; that is, repeated application of the same input has the same effect as one application. This is because the state histories used in composition contain the entire input history, including messages that have been filtered in the upper levels of the tree. For non-idempotent computations, composition results in over-valuation – effectively, every message already filtered above the point of failure will be filtered again resulting in incorrect out-

[3] Data from missing output is contained in the final output stream.

put. A potential approach for non-idempotent computations is discussed in Section 4.

We say filter state for a particular filter is *composable* if the filter state of some other group of processes in the network can be used to regenerate that state, and the generated state is computational consistent with the original state. We observe that composability exists when the output of a filter is a copy of its updated filter state, and the filter function, f, becomes the composition operation.

Theorem 3.1 *Given the definition from Equation (1):* $\{out(CP_i), fs_{p+1}(CP_i)\} \equiv f[IN(CP_i), fs_p(CP_i)]$*, if* $out(CP_i) = fs_{p+1}(CP_i)$*, then* f *can generate compensatory states for* CP_i *using its children's states.*

Proof Let CP_i be a process with children, CP_j and CP_k, and f be a composable filter. By the definition of TBON computations, outputs from children become inputs to parents; in this case $\{out(CP_j), out(CP_k)\} = IN(CP_i)$. Since we assume that the filter state of the children processes are a copy of their output, $out(CP_j) = fs_p(CP_j)$ and $out(CP_k) = fs_p(CP_k)$.

By substitution, we can redefine the filter function of CP_i in terms of the filter state of its children:

$$\{out(CP_i),\ fs_p(CP_i)\} \equiv f[\,\{fs_p(CP_j),\ fs_p(CP_k)\},\ \varnothing\,]$$

That is, by applying f to the filter state of CP_i's children, we can compensate for the filter state of CP_i. We can substitute $\varnothing$, the null value, for the initial state of the filter since the states from the children are aggregates of the entire input history and subsume the parent's filter state.

For the class of filters where filter states are not copies of its output values, composability exists if filter states can be mapped to the output: given a filter f, there exists a mapping g such that, $out(CP_i) = g[\ fs_p(CP_i)\ ]$. A simple extension to the proof of Theorem 3.1, where the inputs to f are the outputs of g applied to the filter state shows this property also leads to composability.

To understand g's derivation in practice, recall that, generally, filter state is an aggregate of the input history, and outputs are the incremental updates caused by new inputs. g could then be the derived from the operations used by the filter to calculate the incremental difference between its previous and current states. For equivalence computations, the outputs are typically set data structures, and g becomes the set difference operator.

To complete recovery, compensatory state must be integrated into the new or preexisting process that adopts the orphaned processes. This is achieved simply by propagating the inputs states to the composition operation as inputs to the filter at the *take-over process*. If the take-over process is a new communication process, then its initial filter state before processing the inputs will be $\varnothing$ as in the discussion above. If a process with prior children is chosen, then it's initial state before composition will be the state it had at the time it was preempted for recovery.

4 Case Study and Discussion

In this section, we solidify the concept of state compensation with a case study of recovery performance and a discussion of some practical matters.

As discussed in Section 2, many compelling algorithms reduce to the integer equivalence computation. Therefore, this computation should be sufficient to demonstrate the generality of our recovery mechanism. In this computation, the TBŌN calculates the list of unique values in the input stream. On each execution, the filter outputs the unique values not previously seen in the input stream. Now consider the scenario shown in Figure 2 in which the root of the given sub-tree has failed. In this case, the computational state at the failed process, along with the in-transit messages from it's children, are lost. Since the integer equivalence filter exhibits all the properties discussed in the Section 3, namely input/output and filter state of the same form, associativity, distributivity and idempotence, we can compose the state of the orphaned children using the integer equivalence filter as the composition operation. The resulting state, $\{2, 3, 6, 7\}$, can now be integrated into the process assuming the role of the failed one, and the resulting TBŌN is guaranteed to produce the same results that would have been produced by the original system had it not experienced a failure.

We briefly discuss the filter that implements a subgraph-folding algorithm (SGFA) [12] to show how our simple case study extends to complex filters. The SGFA merges subgraphs into a single one by calculating equivalent subgraphs and removing redundancies. The filter state at each TBŌN process is the graph resulting from the merger of previous input, and the output is updates to this merged graph caused by new inputs. Just as in integer equivalence, the filter states (graphs) from a failed node's children can be passed through the SGFA filter to compensate for state lost from a parent's failure.

Our recovery process can be implemented as follows: after a failure is detected, the process chosen to be the new parent of the orphaned processes sends a message to the orphaned processes, which signals that they are being adopted by this process. After the new parent/child relationships are established, the children delete any buffered packets that were destined for their original parent (since the effect of these packets are captured by state composition) and propagate the current filter state for all active streams to the new parent as outputs. When these states are read as inputs by the filter in the new parent, the state that compensates for the lost state becomes integrated into the state of the new parent. After propagating a stream's filter state for composition, a child process can resume normal input processing for that stream.

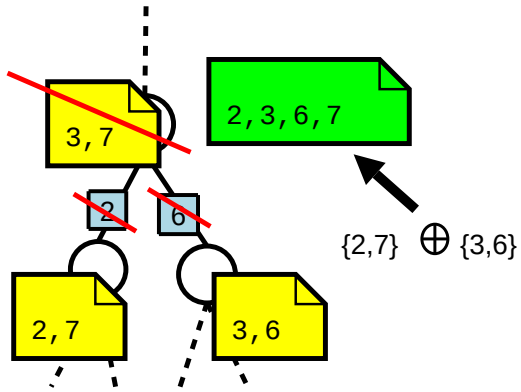

Figure 2: To compensate for the lost filter and channel state after failure of the subtree's root, computationally consistent state obtained by composing the failed process' children's states is used.

We can model recovery performance as:

$$recovery\_latency = fl + s/b + c + o$$

where f is the number of orphans being adopted, l is the latency to send a single message, s is filter state size, b is bandwidth, c is computation time for a single filter execution, and o is a "catch all" for incidental overhead, like calculating the list of orphans and message processing. In practice, f is typically less than 64, so fl the time to send the *NEW_PARENT* messages is low. Even for systems with commodity network interconnections, with filter state size typically on the order of megabytes; s/b, state transmission times, can also expected to be low. For complex filters, incidental overhead should be negligible compared to filter execution time which we expect to dominate recovery. At any rate, it is reasonable to expect very low recovery latencies – a big win given that our overhead during normal execution is nil, and the trade-off is typically quick recovery vs. low runtime overhead.

Discussion Currently, orphaned sibling processes all must be adopted by the same parent. We are studying the notion of "state splitting," which would allow us to redistribute filter state more flexibly – allowing us to assign new parents to orphaned processes more flexibly as well.

We also are investigating other state merging operations to complement state composition. For example, state decomposition operations can generate compensatory states for a failed child by "extracting" the composition of its siblings' states from that of its parent. We believe state extraction will also prove useful to precisely calculate the (effect of) lost messages and compensate for lost channel state in non-idempotent computations . This will avoid over-valuation by only compensating for lost messages, not those that have been filtered above the failure point.

Finally, this work focuses on the reliability of TBON infrastructures; we do not address application process failures outside of the TBON since there may be other application state that needs to be recovered. Our techniques complement traditional approaches like explicit checkpoints that may be needed to tolerate such failures.

Acknowledgments A portion of this work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract W-7405-Eng-48 (UCRL-CONF-222793).

References

[1] Top 500 Supercomputer Sites. (visited July 2006).

[2] D. Arnold, G. Pack, and B. Miller. "Tree-based Computing for Scalable Applications", *11th International Workshop on High-Level Parallel Programming Models and Supportive Environments*, Rhodes, Greece, April 2006.

[3] B. Badrinath and P. Sudame. "Gathercast: The Design and Implementation of a Programmable Aggregation Mechanism for the Internet", *9th International Conference on Computer Communications and Networks*, Las Vegas, NV, October 2000.

[4] S. Balle, B. Brett, C.-P. Chen, and D. LaFrance-Linden. "A New Approach to Parallel Debugger Architecture", *6th International Conference on Applied Parallel Computing. Advanced Scientific Computing (PARA 2002)*, Espoo, Finland, June 2002.

[5] N. Budhiraja, K. Marzullo, F. Schneider, and S. Toueg. "Primary-backup Protocols: Lower Bounds and Optimal Implementations", *3rd IFIP Conference on Dependable Computing for Critical Applications*, Mondello, Sicily, September 1992.

[6] E. Elnozahy, L. Alvisi, Y.-M. Wang, and D. Johnson. "A Survey of Rollback-recovery Protocols in Message-passing Systems", *ACM Computing Surveys* **34** 3, 2002, pp. 375–408.

[7] D. Evensky, A. Gentile, L. Camp, and R. Armstrong. "Lilith: Scalable Execution of User Code for Distributed Computing", *6th IEEE International Symposium on High Performance Distributed Computing (HPDC 97)*, Portland, OR, August 1997.

[8] F. Gärtner. "Fundamentals of Fault-tolerant Distributed Computing in Asynchronous Environments", *ACM Computing Surveys* **31** 1, 1999, pp. 1–26.

[9] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. "Tag: A Tiny Aggregation Service for Ad-hoc Sensor Networks", *5th Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, December 2002.

[10] K. Obraczka. "Multicast Transport Protocols: A Survey and Taxonomy", *IEEE Communications Magazine* **36** 1, January 1998..

[11] P. Roth, D. Arnold, and B. Miller. "MRNet: A Software-based Multicast/Reduction Network for Scalable Tools", *SC 2003*, Phoenix, AZ, November 2003.

[12] P. Roth and B. Miller. "The Distributed Performance Consultant and the Sub-graph Folding Algorithm: On-line Automated Performance Diagnosis on Thousands of Processes", *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '06)*, New York, NY, March 2006.

[13] F. Sacerdoti, M. Katz, M. Massie, and D. Culler. "Wide Area Cluster Monitoring with Ganglia", *IEEE International Conference on Cluster Computing*, Hong Kong, September 2003.

[14] M. Sottile and R. Minnich. "Supermon: A High-speed Cluster Monitoring System", *IEEE International Conference on Cluster Computing*, Chicago, IL, September 2002.

[15] R. Strom and S. Yemini. "Optimistic Recovery in Distributed Systems", *ACM Transactions on Computer Systems* **3** 3, 1985.