RELIABLE, SCALABLE TREE-BASED OVERLAY NETWORKS

by

Dorian Cecil Arnold

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2008

To my dearest Jay, Denice and DJ.

Without you, I have nothing. Without you, I am nothing.

ACKNOWLEDGMENTS

Ultimately, our Creator is responsible for all.

I was fortunate to have an excellent and strong support system throughout this process. With great pleasure and extreme gratitude, I thank you all.

I am deeply grateful to Barton Miller, my advisor. Your reputation for high standards and excellence preceded you. You did not disappoint. Under your mentorship, I have matured tremendously as a researcher, a thinker and a person. You said that the hard work would pay off and that my dissertation would make me proud. You were right. The very first feedback on my dissertation that I received was from a (soon to be) fellow Ph.D. whose own dissertation won the ACM Doctoral Dissertation Award; he was impressed by the clarity of expression and the superb quality of the writing. I responded that my advisor would settle for nothing less. You also have taught me to settle for nothing less.

I thank my proposal committee, Suman Banerjee and Miron Livny, and my dissertation committee, Ben Liblit, Jignesh Patel, Michael Swift and Paul Wilson. They provided helpful feedback that has improved the quality of my dissertation and were very flexible in accommodating our scheduling constraints and tolerating an 8:00 a.m. defense.

I am forever indebted to past and present members of the Paradyn Project and my friends from the Computer Sciences department, particularly, Will Benton, Drew Bernat, Mike Brim, Laune Harris, Alex Mirgorodskiy, Matt Legendre, Nate Rosenblum, Kevin Roundy, Phil Roth and Vic Zandy. Over the years, these fine folks have been invaluable in all aspects of the graduate school experience from the technical to the social. Phil was an excellent colleague as we collaborated on the early phases of MRNet during his tenure at the University of Wisconsin. And Will, your dear friendship made the years in Madison seem like, well, less years ☺. It is fitting that we have reached this milestone together, and I hope our families remain close despite any physical distance.

I had the great fortune of working with Dong Ahn, Bronis de Supinski, Greg Lee and Martin Schulz from the Lawrence Livermore National Laboratory. That opportunity has been both fun and productive, and I look forward to maintaining our collaboration. Also, most of the results presented in this work would not be possible without the computer resources and support I received from the Lab.

I thank Jack Dongarra and James Plank who provided me with a most excellent start in this world of Computer Science research and who have continued to be valuable sources of mentorship and support.

I thank the Intel Corporation for giving me a graduate fellowship that provided two years of financial support for graduate school.

It Takes A Village

Many have walked alongside me at some point on this journey – my ever loving grandmothers, Eileen and Gladys, my aunts in Belize and the U.S., and a collection of family and friends scattered all over the globe. My parents-in-law, Oly and Franklyn, have welcomed me into their family and have showered us with unending generosities.

Big up to all a mi bwai dem fram yaad. Unnu mek ah feel lyk Belize neva deh so far aweh. Espeshali Stewart Cruz, Francis Perez, Elroy Shaw and Kareem Usher. Tanks fu di regula canvasayshan, di advise pon skool, famli, life in jenaral an iiven di canvasayshan dem bout nutin at all.

I am who I am because of my parents, Carolyn and Cecil. I thank you for your endless labour and sacrifices through the years. There’s nothing you would not do for us, and we love you for that. I hope that I now can be closer to home both virtually and physically. Mom, I still owe you a car for the one I wrecked – I have a job now, so maybe soon ☺. And to my Aunt Elaine, you opened your heart and home to me and treated me like a son (scoldings for coming home late and all ☺). I am forever in your debt.

Finally, I would like to thank my wonderful family, Jay, Denice and DJ, for their unconditional love and support throughout this long journey. They say having a wife and kids while going to school places an unbearable strain on an already arduous process. I disagree. I have succeeded because of my wife and kids, not despite them. When times were tough, knowing that there were people and things much more important than my schooling and career always kept things in proper perspective. I could not have done this without you guys. Denice and DJ, you probably do not remember repeating “professa” a while back when we were talking about what I would like to become. That chant often echoed in my head and provided that extra push when I needed it. Jay, you are a very special woman, pure of heart. Never did I experience the cliché “spouse of graduate student” syndrome. You were always patient, and never did I doubt that you would support me no matter what. I love you forever with all my heart.

RELIABLE, SCALABLE TREE-BASED OVERLAY NETWORKS

Dorian Cecil Arnold

Under the supervision of Professor Barton P. Miller

At the University of Wisconsin-Madison

As high performance computing (HPC) systems continue to increase in size, reliable and scalable computational models become critical. Tree-based overlay networks (TBŌNs) help to address scalability by providing scalable data multicast, data gather, and data aggregation services. In this dissertation, we address the reliability challenges of TBŌN-based HPC tools and applications.

We exploit the characteristics of many TBŌN computations to develop a new failure recovery model, state compensation, that uses:

- inherently redundant information from processes that survive failures to compensate for information lost due to failures,
- weak data consistency to relax the constraints of the recovery mechanisms, and
- protocols that allow processes to recover from failures independently.

State compensation requires no additional computational, network or storage resources in the absence of failures. When failures do occur, a small subset of TBŌN processes participate in failure recovery, so failure recovery is scalable.

We developed a formal specification of our data aggregation model that allowed us to validate our failure recovery mechanisms and identify their requirements and limitations. Generally, state compensation requires that data aggregation operations be commutative and associative. Our primary compensation mechanism requires that the data aggregation operation be idempotent. Our second compensation mechanism addresses non-idempotent data aggregation operations using more complex recovery mechanisms.

We studied tree reconfiguration algorithms for high performance TBŌNs, focusing on the algorithms' execution times and the costs of managing the TBŌN process information needed by the reconfiguration algorithms. We also considered the data aggregation latency of the resulting configurations, and concluded that this should be the primary consideration for TBŌNs with up to one million application processes. We recommend an algorithm that considers all TBŌN processes but restricts increases in tree height, since height increases can have a significant negative impact on data aggregation performance.

Also, we implemented our primary state compensation mechanisms. Our experiments with this framework confirm that for TBŌNs that can support millions of application processes, state compensation can yield low failure recovery latencies and inconsequential application perturbation.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

Reliable, scalable computing models have become critical as the sizes and complexities of high performance computing (HPC) systems continue to increase. In HPC environments, tree-based overlay networks (TBŌNs) have been shown to provide a powerful computation model for tool and application scalability [7, 9, 33, 58, 76, 77, 79, 87]. A TBŌN is a network of hierarchically organized processes that leverages the logarithmic scaling properties of the tree organization to provide scalable data multicast, data gather, and in-network aggregation. Our thesis is that by exploiting the characteristics of many TBŌN computations, we can enable new failure recovery models that are scalable and low overhead with little application perturbation. This dissertation describes and evaluates a novel TBŌN failure recovery model and a related set of lightweight mechanisms for constructing robust HPC tools and applications.

1.1 Reliability in Large Scale Systems

Large numbers of processors account for a significant portion of the increased capabilities of HPC systems. In fact, energy and heat concerns have led to solutions, like IBM's Blue Gene series [14], which use less capable processors that run at lower clock rates, but many thousands of them. Today, BlueGene/L (BG/L) at the Lawrence Livermore National Laboratory is comprised of 106,496 nodes and 212,992 processors. Today one can purchase a 256 rack BlueGene/P system with 262,144 nodes and 1,048,576 processors, and BlueGene/Q systems (to be available around 2010) are expected to have even more processors per node. The first million processor systems are slated for service availability around 2010–2012. Data from the Top500 [97] show that the total number of processors comprising all 500 entries has an average growth factor of 1.29 per year [91].

In addition to increasing sizes, HPC systems are also increasing in component complexity, which leads to individual components that are more failure prone. Furthermore, the increased sizes lead to systems with low mean time between failures (MTBF). For a system of identical components, system MTBF is inversely proportional to system size:

$$\text{System MTBF} = \frac{\text{Component MTBF}}{N}$$

where N is the number of components in the system.

Schroeder and Gibson conducted a reliability study [83] based on failure data for 22 high-performance systems at the Los Alamos National Laboratory (LANL) collected from 1996–2005 [57, 82]. Indeed, a principle finding was that system failure rates depend mostly on system size, particularly, the number of processor chips in the system. In related work [38], they predict that if HPC systems grow in size by a factor of two every 18 to 30 months, expected system MTBF for the biggest machines on the Top 500 lists will fall below 10 minutes in the next decade.

Table 1.1 shows reliability data for several HPC systems. BG/L, ASC Purple, and ASCI White are at the Lawrence Livermore National Laboratory; Bassi, Franklin, Jacquard, PDSF and Seaborg are at the National Energy Research Scientific Computing Center (NERSC), and the unnamed systems are the LANL clusters[1] from the Schroeder and Gibson study[2]. BG/L failure statistics report on its previous configuration with 65,536 nodes [94]. Purple and White failure statistics are from anecdotal data, including a presentation on ASCI White [85], and the data for the NERSC systems reveal their 2007 failure statistics [65]. The last column of Table 1.1 is the projected system MTBF for 10^6 processor versions of these systems. These projections corroborate Schroeder and Gibson's projections of multiple failures per day, if not hour.

In general there are inherent trade-offs between performance and reliability, particularly in distributed systems. Scalability infers simple, low-overhead, decentralized mechanisms, whereas the global coordination and data consistency often required for

[1]While most of the largest HPC systems are custom MPPs, commodity clusters remain significant: on the current Top500 list two of the top five entries, including #1, and 80% of the entire list are clusters.

[2]ASCI White and Seaborg were decommissioned in 2006 and 2008, respectively.

System	**Procs.**	**System MTBF (days)**	**Processor MTBF (years)**	**MTBF: 10^6 Procs.**
BG/L (IBM BlueGene)	131,072	6.23	2237.20	19h 36m
Seaborg (IBM SP)	6,080	14.59	243.03	2h 8m
Franklin (Cray XT4)	19,320	1.86	98.45	52m
White (IBM SP)	8,192	2.13	47.69	25m
Purple (IBM Power5)	12,256	1.25	41.97	22m
Jacquard (Linux cluster)	712	16.02	31.25	16m
Bassi (IBM p575 cluster)	888	11.55	28.10	15m
PDSF (Linux cluster)	550	9.05	13.64	7m
Cluster 1	6,152	0.45	7.58	4m
Cluster 2	544	5.06	7.54	4m
Cluster 3	1,024	2.63	7.38	4m
Cluster 4	512	3.35	4.70	2m
Cluster 5	2,048	0.78	4.38	2m
Cluster 6	128	11.86	4.16	2m
Cluster 7	4,096	0.32	3.59	2m
Cluster 8	4,096	0.31	3.48	2m
Cluster 9	512	2.43	3.41	2m
Cluster 10	2,048	0.55	3.09	2m
Cluster 11	328	3.40	3.06	2m
Cluster 12	256	4.11	2.88	2m

Table 1.1 HPC Failure Statistics: Reported and projected failure rates for several HPC systems from the Lawrence Livermore National Laboratory, the Los Alamos National Laboratory, and the National Energy Research Scientific Computing Center.

distributed system reliability tend toward complex, high-overhead, centralized mechanisms. We further identify the challenges of reliability in large scale systems as well as the design goals and constraints that fall out from these challenges:

- *Replication*: There is no reliability without redundancy: systems that tolerate failures must employ either space redundancy (replicating the results of prior computations) or time redundancy (replicating prior computations) [37]. The costs of such replication schemes increase linearly with system scale. A scalable failure recovery model must minimize overhead by distributing replication costs, or by avoiding or minimizing explicit replication mechanisms altogether wherever possible.

- *Data Consistency*: Particularly in the presence of failures, components of a distributed systems must maintain a globally consistent view of data. In other words, the system must assert some guarantees of how failures will affect the values of data it manages or outputs. Strong data consistency guarantees that each system component will always have the same view of managed data or that (for a deterministic computation) output data will be the same regardless of failure conditions. However, the mechanisms necessary to maintain such strong assertions typically cause high overhead. A scalable failure recovery model will leverage weaker, more relaxed consistency models where possible.

- *Coordination*: Distributed systems must often employ mechanisms to achieve system-wide objectives. For example, overlay networks need to determine process configurations that yield good overall application performance. However, as with strong data consistency protocols, inter-process coordination protocols to accomplish global conditions are costly. Scalable designs will avoid global mechanisms in favor of localized ones that achieve similar results.

- *Information Dissemination*: It is typically necessary to propagate system status information globally. For example, as system configuration changes in response to stimuli like failure and recovery events, components must be notified of these changes since they impact system functionality. This information must be broadcast efficiently to all components. Efficient dissemination mechanisms will keep the overhead of such system-level activity low to minimize application perturbation.

1.2 TBŌN-based Applications

Tree-based computing models have long been employed for scalable computing. As early as 1980, Ladner and Fischer observed that hierarchical decomposition in the form of parallel prefix computations can be used for efficient processing [49]. Today, TBŌNs are used for scalable multicast [67]; data aggregation services [7, 9, 33, 76]; distributed debugging, performance and monitoring tools [77, 79, 87]; information management systems [75, 102]; stream processing [8]; and mobile ad hoc networks (MANETs) [58, 104].

TBŌNs are used for many types of simple and complex data aggregation operations. For example, the Ganglia cluster monitoring tool [79] uses its *monitoring tree* to compute summary statistics (sums, averages, upper/lower bounds, etc.) of cluster node information. Ygdrasil [9] uses an *aggregator tree* to condense identical or nearly identical textual output from tools like debuggers. TAG [58] uses an SQL interface to query and aggregate data in sensor network environments. Our own TBŌN prototype, MRNet (described in Chapter 4), has been used for complex computations like clock synchronization [76], equivalence classifications and time-aligned data aggregation [76], scalable performance analysis [63, 77], and scalable group file operations [18].

MapReduce [28] is another programming model for scalable data aggregation. In this model users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key. As shown in Figure 1.1, MapReduce is an instance of the general TBŌN computational model where the TBŌN has two intermediate levels. The last level of TBŌN processes, the level furthest from the root, executes the map function on input data and passes the intermediate results to the first level, which executes the reduce function.

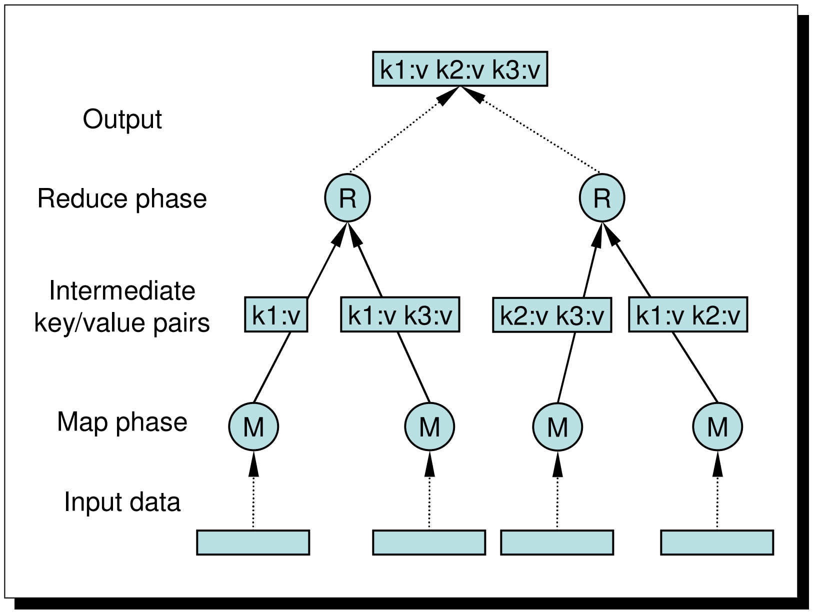


Figure 1.1 MapReduce: The map function, *M*, generates intermediate key/value pairs from the input data. The reduce function, *R*, merges all intermediate values associated with the same key.

1.3 Scalable TBŌN Fault Tolerance

Our goal is to design failure recovery models and mechanisms that allow TBŌN computations to retain their efficient, scalable performance characteristics. We target HPC applications that require high throughput, low latency communication of possibly large amounts of data. Lastly, we target future petascale and exascale systems that will support distributed applications with millions of components. Our central thesis is that we can exploit the structure and properties of TBŌN-based computations to employ recovery models that utilize no (or extremely low) additional computational resources in the absence of failures but are responsive and cause little application perturbation when failures do occur.

Our approach is motivated by three fundamental observations:

1. There exist inherent information redundancies within the computational structure of *stateful* TBŌN-based data aggregations: as information is propagated from the leaves of the tree toward the root, aggregation state, which generally encapsulates previously processed information (input history), is replicated at successive levels in the tree.

2. Maintaining strong data consistency in distributed systems often leads to complex mechanisms with large overheads. More importantly, many useful computations do not require strong data consistency.

3. Recovery models that require process coordination or global consensus are inherently non-scalable.

In this dissertation, we use these observations to design, implement and evaluate a scalable failure recovery model for TBŌNs. We use the inherent TBŌN information redundancies to recover lost state without explicit replication mechanisms. We use weak data consistency models to relax the constraints of our failure recovery mechanisms, such as our tree reconstruction and information dissemination protocols. Finally, our recovery model leverages localized protocols that allow components to operate in a completely independent fashion.

1.4 Contributions

This dissertation makes several contributions in reliable, scalable data aggregation:

- A formal specification of the data aggregation model that serves as the basis for the analyses and correctness proofs of our recovery model and its properties;
- *State Compensation*, a collection of scalable algorithms for TBŌN state recovery;
- New tree reconstruction algorithms; and
- An implementation of these concepts to demonstrate their performance and practical application.

We preview these contributions here and explore them in detail in subsequent chapters.

A Formal Aggregation Model: We develop a formal specification of the TBŌN computational model and its properties and use the specification to validate our recovery model. This formalization allows us to reason about the assumptions of our recovery model as well as its constraints and limitations. Further, the specification informs and directs the implementation of the recovery model.

State Compensation: The primary contribution of this dissertation is a novel state recovery strategy for TBŌNs. We call this new model *state compensation* because we use process state that survives process, host, or network failure(s) to compensate for other process and channel state that we may have lost due to the failure(s). State compensation leverages the inherent information redundancies found in many TBŌN applications to avoid explicit data replication, which limits the scalability of contemporary approaches as discussed in Chapter 2.

State compensation also leverages a form of weak data consistency called *equivalent recovery* [43], where post-failure output is equivalent, but not identical, to the output of a non-failed execution. However, previous failure recovery models based on this form of data consistency model still rely on explicit replication. We demonstrate the use of weak consistency models to completely avoid explicit data replication.

Tree Reconstruction Algorithms: When failures occur, disconnected subtrees must be reconnected back into the main tree trunk. We developed and evaluated several lightweight tree reconstruction algorithms that are completely localized: orphaned nodes make reconnection decisions using only locally available information. We compare their computational requirements and the quality of the reconstructed trees using depth and maximum fan-out, which determine the trees' communication latency and band-width, and the standard deviation of the fan-out, which measures the trees' balance. Our results show a trade-off between algorithm performance and scalability and the quality of the reconstructed trees. We have evaluated the algorithms on trees with more than one million nodes, and for such trees, the algorithms that produce the best trees are efficient enough in computational overhead and space requirements to be used in practice.

Reliable, Scalable TBŌN Implementation: We develop an implementation of our recovery model using the MRNet TBŌN prototype [76]. The implementation includes all components of our failure recovery model including our state compensation, information dissemination, and tree reconstruction algorithms. Using this prototype, we demonstrate that a TBŌN structure that readily supports millions of processes can recover from failures in milliseconds. We also demonstrate that the failures cause little application perturbation. The prototype is now available for use by real world applications and tools.

1.5 Dissertation Organization

The remainder of this dissertation is organized as follows: In Chapter 2, we present background and related research in data consistency models, distributed systems fault-tolerance, scalable information dissemination, and tree reconstruction algorithms. In Chapter 3, we describe the general approach to TBŌN-based computing and present our first contribution, a formal specification of the TBŌN computational model. In Chapter 4 we present a brief overview of the MRNet TBŌN prototype. In Chapter 5, we present an MRNet case study that demonstrates the scalability of the TBŌN paradigm. This application is additionally relevant because it exhibits the properties required by our failure recovery model.

The remaining chapters present the core contributions of this dissertation. In Chapter 6, after describing our failure, data consistency and failure recovery models, we present, state compensation, a scalable model for recovering lost TBŌN state. We describe the fundamental TBŌN properties upon which state compensation is based and formally show that state compensation preserves computational semantics in the presence of failures. In Chapter 7 we present a study of tree reconstruction algorithms. In Chapter 8, we describe our initial MRNet-based prototype implementation of our scalable failure recovery model and present the results of our empirical analysis. We conclude with a summary of our research contributions and future directions in Chapter 9.

Chapter 2

Background and Related Work

We describe previous research related to reliable data aggregation and tree reconstruction algorithms. First, we survey fault tolerance approaches that may be applied generally to distributed systems, namely *hot backup* and *rollback recovery protocols*. Then, we discuss existing techniques designed specifically for reliable data aggregation. Finally, we examine previous work in tree reconstruction techniques.

The key distinction between existing failure recovery models and the one we propose for TBŌN aggregation is that the former all use explicit data replication protocols, which consume computational, network or storage resources during normal operation. Additionally, the general fault-tolerance approaches do not scale well and are not suitable for extremely large scale environments. Finally, the aggregation specific approaches either apply to only specific operations like *sum* or *average*, or are suitable for only computations on small amounts of data that can tolerate high latencies. In contrast, our failure recovery model leverages information redundancies inherent to the computation eliminating explicit replication or coordination protocols. We therefore place no additional burden on the system for explicit replication, while being suitable for large classes of TBŌN-based aggregation operations.

2.1 Hot Backup Protocols

In hot backup protocols, primary components are backed up by replicas that can immediately provide the same service as the primaries, should they fail. We survey software-based hot backup protocols, in which distributed systems are comprised of the *primary processes* necessary to implement the system's functionality and *backup processes* that replicate the primary processes' functionality for reliablity. Hot backup protocols can be divided into two categories: *inactive backup* and *active backup* protocols.

2.1.1 Inactive Backup Protocols

In inactive backup (also called *inactive standby*, *passive standby* or *primary-backup*) protocols, backup processes only interact with their primaries or other system components solely to maintain some level of synchrony with their primaries. Alsberg and Day described an early single primary, multiple backup protocol [3] in which a primary process and its backup execute the same computation: input messages received by the primary process are propagated to each of its backups in the same order that the primary received those messages. To guarantee that the system is recoverable, a primary process does not acknowledge receipt of an input message until it has been propagated to at least one of its backups. If the primary process fails, a backup process is elected as new primary. If a backup process fails, references to it are removed from the surviving processes.

Bartlett developed one of the first implementations of an inactive backup protocol for the Tandem computer system [10]. Server *process pairs*, comprised of a primary process and its backup, managed the system's I/O devices. As in Alsberg and Day's protocol, client requests were propagated from the primary to its backup so that if the primary failed, its backup had the necessary information to assume control of the relevant device. The system used *request sequence numbers* to assure that non-idempotent operations were performed exactly once.

Concurrently, Borg, Baumbach, and Glazer developed a fault-tolerant message system [16] and Powell and Presotto developed *Publishing* [73]. Borg et al used kernel services periodically to synchronize a primary process with its remote backup; between synchronization events, messages to the primary were logged for its backup. In contrast, Powell and Presotto used a central *recorder process* to which checkpoints of primary processes were sent. In addition to saving these checkpoints, the recorder process passively snooped and logged all network traffic destined to primary processes. In both systems, when a primary process failed, its backup replaced it. The system delivered to the backup all messages that the primary had received after its most recent synchronization event or checkpoint and suppressed any output that the primary had already transmitted

The last inactive backup protocol we survey is Pronto [70], which applied the inactive backup scheme to replicate entire databases. Pronto orchestrated transaction processing amongst multiple, standard databases to provide the image of a single, highly-available database. The details for replica coordination were similar to those above: transactions were broadcast to all database replicas. Upon failure, one of the replicas was promoted to be the new primary database server.

2.1.2 Active Backup Protocols

In active backup (also called *active standby*) protocols, all replicas are used to service client requests. Schneider introduced the active backup concept with a proposal to implement fault-tolerant services using state machines [81]. He demonstrated that the correctness of system output can be guaranteed if all non-faulty replicas receive and process the same sequence of inputs in the same relative order. Each replica concurrently processes input messages; to tolerate fail-stop failures, any replica's output can be chosen since no replica produces faulty output under this failure model. To tolerate Byzantine failures, majority consensus of the replicas' output is necessary.

The Delta-4 distributed computing system [24] supported an active replication model where the system's software components are replicated and requests for software services are executed concurrently by all replicas. Delta-4 implemented an *inter replica protocol* based on Schneider's state machine approach to coordinate replica consistency. Zhou, Chen and Li used the virtual memory-mapped communication (VMMC) model to efficiently mirror process address spaces across remote memories [108]. They proposed two protocols for replica coordination an *automatic update* and a *deliberate update protocol*. In the automatic update protocol, processes map virtual memory to data buffers that are imported by their remote replicas. Custom network interfaces are used to automatically update the corresponding remote buffers when the process writes to its virtual memory.

The deliberate update protocol is similar to previously described approaches and uses explicit messages for replica coordination.

2.1.3 Discussion of Hot Backup Protocols

Hot backup protocols have low failure recovery latencies since backups are kept in a (near) ready state. These protocols, particularly inactive backup protocols, employ simple coordination mechanisms that are straightforward to implement. Using multiple backups per primary, hot backup protocols can support Byzantine failures in addition to fail-stop failures, though the protocols we surveyed only support fail-stop failures.

The main drawback of hot backup protocols is their overhead during normal execution. Even in single primary, single backup schemes, these protocols suffer an overhead of 100% – likely prohibitive for all but the most mission critical tasks. Additionally, replica coordination can delay each input communication event of the primary processes. This additional latency may negate the potential service speedups of the active backup model.

2.2 Rollback Recovery Protocols

In *rollback* (or *backwards-error*) *recovery protocols* [30], during normal operation, process or communication state is periodically recorded to stable devices that survive all tolerated failures. When a failure occurs, the system rolls back to the point represented by its most recent stored state – thereby reducing the amount of lost computation. *Checkpoint-based protocols* and *log-based protocols* are the main variants of distributed rollback-recovery.

2.2.1 Checkpoint-based Rollback Recovery

In checkpoint-based protocols, process state is periodically *checkpointed* to stable storage. Process state is all state necessary to run a process correctly including its memory and register state. If a process failure is detected, the failed process' most recent checkpoint is used to restore (a new incarnation of) the failed process to the intermediate state saved in the checkpoint. Several optimizations have been proposed to improve basic checkpointing including:

- *Incremental checkpointing* [23, 31, 54, 72]: the operating system's memory page protection facilities are used to detect and save only pages that have been updated between consecutive checkpoints.

- *Forked checkpointing* [31, 35, 53, 55, 54, 69]: the application process forks a *checkpointing process* allowing the original process to continue while the forked process concurrently commits the checkpoint state to stable storage. If the *fork* system call implements *copy-on-write* semantics, both processes efficiently share the same address space until the original process updates a memory segment at which point a copy is made so that the checkpointing process has a copy of the memory state at the time the checkpoint was initiated.

- *Remote checkpointing* [90, 106]: Remote checkpointing leverages network resources to save checkpoints to remote checkpoint servers providing performance gains in environments where I/O bandwidth to the network is more abundant than that to local storage devices. Additionally, remotely stored checkpoints allow systems to survive non-transient node failures.

Distributed checkpointing protocols save and restore *global checkpoints* comprised of a set of local checkpoints, one from each process in the system. Inter-process dependencies (which result from inter-process communication) complicate checkpointing and lead to two consequences: first, failures may force non-failed processes to rollback to previous checkpoints; this is called *rollback propagation*. Second, they constrain which global checkpoints reflect a *consistent* and useful system state. Intuitively, a consistent state is one that may occur during the correct execution of a computation [30]. In Figure 2.1; the global state $\{s_0, s_1\}$ is *inconsistent* because it reflects an *orphaned message*, m_0 – which is orphaned because s_1 reflects m_0 as received by process P_1, but s_0 reflects m_0 as not having been sent by P_0. In other words, a *consistent system state* is one with no orphaned messages [20]. A transitless system state is one that reflects no in-transit messages [42]: the global state, $\{s_2, s_3\}$, is not transitless because it reflects message m_1 as in transit [1]. Finally, a strongly consistent state, for example, $\{s_4, s_5\}$, is one with no orphaned or in-transit messages [42]. In principle, a global state only has to be consistent, not strongly consistent, to be useful. However, in practice most checkpoint protocols do not leverage connection migration techniques, which re-route in-transit messages destined for a failed process to its new incarnation. Thus, to provide reliable message delivery, most rollback

[1]Of course, in-transit messages occur in any practical distributed system with non-zero message latencies and do not reflect an inconsistency.

recovery protocols rely on strongly consistent global states. (There do exist at least two checkpointing protocols that incorporate connection migration [68, 105].)

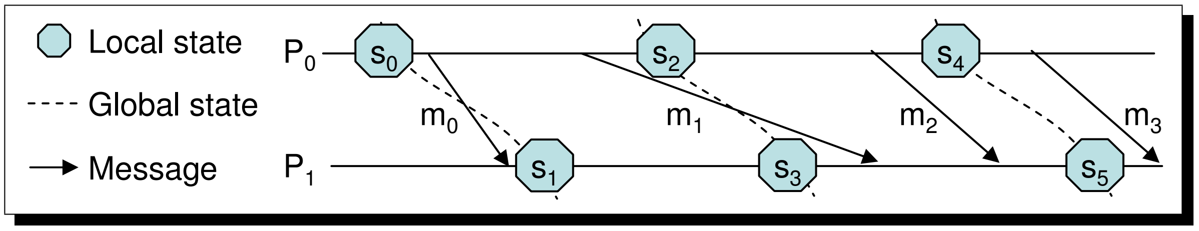


Figure 2.1 Global State Consistency: $\{s_0, s_1\}$ is inconsistent because m_0 is orphaned. $\{s_2, s_3\}$ is not transitless because m_1 is in transit. $\{s_3, s_4\}$ is strongly consistent – consistent and transitless.

There are three styles of distributed checkpointing: *uncoordinated*, *coordinated* and *communication-induced*. In uncoordinated checkpoint protocols [12, 99], processes independently checkpoint their local state. There is no guarantee that any single checkpoint will comprise a globally consistent state, so to minimize rollback propagation, each process must maintain multiple checkpoints. During the recovery phase, the system computes the set of local states that comprise the most recent consistent global state. In the worst case, no such set exists and rollback propagation leads to the *domino effect* [74], where the computation is forced to rollback to its initial state, losing all prior work.

Coordinated checkpointing limits rollback propagation and eliminates the domino effect by preempting all processes in the system at the time of a checkpoint. The processes then coordinate to ensure that the local checkpoints comprise a consistent global state; therefore, each process only needs to maintain its single, most recent checkpoint. Such protocols can be implemented simply by blocking all process communications when the checkpoint protocol executes [93] or using more complex non-blocking algorithms [20, 31, 50, 86]. There also have been proposals to coordinate checkpoints using synchronized clocks as opposed to inter-process communication [26, 96].

Communication-induced checkpointing is a hybrid of coordinated and uncoordinated checkpointing. Processes are allowed to take independent checkpoints but also must take *forced* checkpoints based on communication patterns – particularly to avoid *useless* checkpoints that may never be a part of any consistent global state. No explicit checkpoint protocol messages are used; instead, they piggyback these messages on application messages. The two main approaches for determining when forced checkpoints must be taken are model-based protocols [66, 78, 100], which prevent patterns of checkpointing and communication that create useless checkpoints, and index-based protocols, which use logical clocks [51] to timestamp (and force) checkpoints in such a way that no useless checkpoints are created [17, 41].

2.2.2 Log-based Rollback Recovery

Checkpoint protocols require that all processes, even non-failed ones, rollback to their most recent globally consistent state. Log-based recovery (or *message logging*) protocols avoid rolling back non-failed processes by recovering a failed process to pre-failure its state by logging its external interactions for replay, as necessary, after a failure [16, 73, 92]. Log-based recovery relies on *piecewise determinism*, which assumes that all non-deterministic events can be identified, logged and replayed [92]. Log-based recovery typically complements other reliability mechanisms that periodically save process state to limit rollback after a failure. For example, Borg, Baumbach, and Glazer's message system [16] and Powell and Presotto's Publishing system [73] logged all message transmissions during failure-free execution. Failed processes were recovered using fail-over or checkpointing mechanisms, respectively, and the message logs were used to replay messages received and suppress messages sent by the failed process since the last time the failed process was synchronized with its replica or checkpointed.

2.2.3 Discussion of Rollback Recovery

Rollback recovery protocols are perhaps the most well-known techniques for fault tolerance in distributed systems, particularly because they generally apply to any distributed system or application. Researchers have studied the characteristics of these protocols for over three decades and have established many useful theories about their behavior, consistency models and recovery guarantees.

However, performance drawbacks make rollback recovery protocols in their current form unsuitable for large scale computing environments with hundreds of thousands of components or more. As the number of computational nodes an application uses increase, so does the application's global checkpoint overhead. At the same time, the increased MTTF that results from the increased number of nodes suggests that an application should take checkpoints more frequently to minimize work loss. Recent studies [32, 84] have suggested that the combination of these factors will result in poor application utilization. The studies project that large applications running on near future systems that use rollback recovery fault tolerance will require resource allocations devoted to reliability (a departure from the current practice of "borrowing" application cycles) or spend most of their time writing checkpoints or recovering lost work.

2.3 Reliable Data Aggregation

Researchers have proposed mechanisms specifically designed for reliable data aggregation in a variety of domains including stream processing engines (SPE), distributed information management systems (DIMS), and mobile ad hoc networks (MANETs). Some of the approaches leverage previously described reliability mechanisms such as hot backup and checkpointing protocols, while others use application specific protocols.

2.3.1 Stream Processing Engines

SPEs are similar to TBŌN environments: both SPEs and TBŌNs aggregate waves of input data by propagating them through a system of processes and produce waves of output. The primary difference is that SPEs may organize these processes into directed acyclic graph topologies whereas TBŌNs use tree topologies: TBŌNs are a sub-class of SPEs. The Borealis SPE [8] uses an inactive backup protocol to replicate its query processing nodes. To maintain consistency amongst replicas, Borealis uses a data serializing operators that establish the same input ordering at all replicas. If a node detects that one of its input sources have failed and it cannot find a replica for the failed node, it can either delay processing until the failed node is repaired or produce *tentative* (as opposed to *stable*) output. Once the network is healed, any tentative output must be reconciled. Reconciliation is executed by rolling back a node to a known non-tentative state and replaying stable output that updates previously tentative ones. Rollback can be checkpoint based or done by logging tentative inputs and using aggregation dependent routines to *undo* them once stable versions are available.

Hwang, Balazinska, Rasin, Centintemel, Stonebraker and Zdonik [43] study three recovery guarantees for SPEs: *gap recovery*, *rollback recovery* [2], and *precise recovery*. In gap recovery, the SPE is repaired after failures, but no attempt is made to recover lost data, and output may contain gaps of missing data. In rollback recovery, simple active backup, inactive backup, and message logging protocols are used. In this case, their use of these protocols assumes that the aggregation operations are idempotent, and output produced after a failure is *equivalent*, but not necessarily equal, to the output of an execution without failure. They define a computation that has experienced a failure as producing output equivalent to that of a non-failed instance of the same computation if in the former case, all input data are processed by non-failed processes at least once despite the failures. The output may contain duplicates due to processing input data more than once or may contain different data values due to processing input data in a different order or grouping. For precise recovery, they add order preserving and duplicate suppressing mechanisms to their initial active backup, inactive backup and message logging techniques.

[2]Hwang et al. define rollback recovery as a data consistency type; rollback recovery traditionally is defined to be a failure recovery protocol, as described earlier.

2.3.2 Distributed Information Management Systems

DIMS [75, 102] are designed to manage and aggregate dynamically changing information about large scale networked environments. Typically, clients can access the managed information from any node with queries for summaries. To support such operations, DIMS employ protocols to disseminate all known information in aggregate form to all members of the system.

As in TBŌNs, DIMS often employ hierarchical structures for scalability [40, 75, 102]. In Astrolabe [75] and in the aggregation approach proposed by Gupta, van Renesse and Birman [40], *gossiping* is used for robustness. Both approaches organize nodes into disjoint clusters, and impose a hierarchy by iteratively forming larger disjoint clusters of clusters. Each cluster has an elected leader: at each nesting level, the leader is chosen from amongst the leaders of the enclosed clusters. This scenario is shown in Figure 2.2 for a two-level hierarchy of 16 nodes. For each cluster, the node with the lowest identifier is the cluster leader. In Astrolabe, nodes replicate aggregates of attribute information by periodically gossiping with a node chosen from all other nodes in the system. The hierarchy is used to determine the level of information exchanged during a gossip: nodes in the same un-nested cluster exchange all known attribute information for that cluster; nodes in different clusters exchange attribute information for their least common ancestor.

As in TBŌNs, Gupta et al's approach produces the global aggregate at the root of the hierarchy. Their algorithm executes in h phases, where h is the height of the tree. For each $i : 1 \leq i \leq h$, the roots of the height $i - 1$ subtrees gossip amongst themselves to

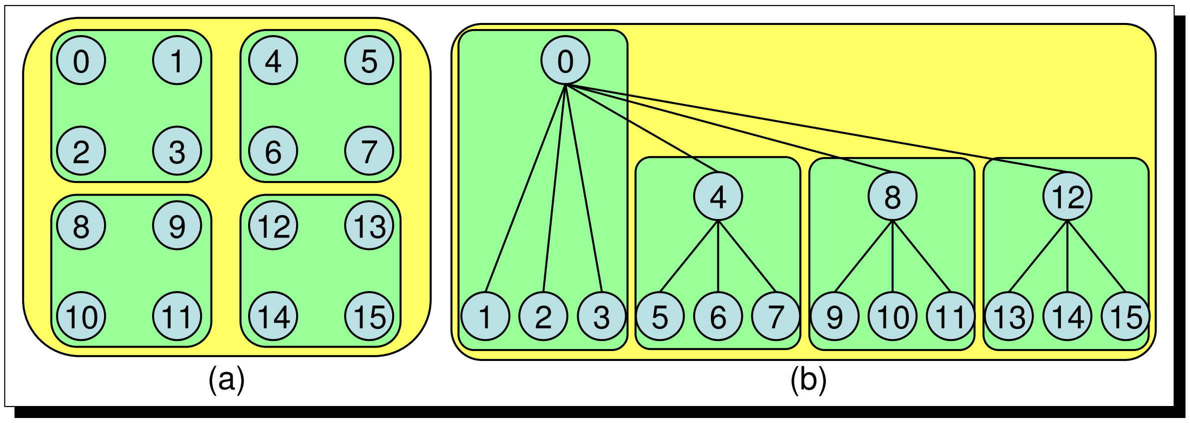


Figure 2.2 Hierarchical Clusters: (a) 16 nodes divided into a 2-level cluster of clusters. (b) The hierarchical representation: for each cluster, the node with the lowest identifier is elected leader.

disseminate their current aggregate estimates. At the end of phase i, the roots of the height i subtrees compute a new estimate based on aggregates received during that phase. At the end of phase h, the aggregate estimate at the root of the tree is the aggregate estimate of the entire system.

SDIMS [102] also uses a tree to organize its managed components. To tolerate failures (as well as to improve query response times), SDIMS uses an explicit replication protocol. Each node maintains raw and summary attributes for its children, as well as summary attributes scattered by its ancestors and descendants – users can control how far up or down the tree a node's summary information is replicated. Tree reorganization due to failures can lead to incorrect summary information. To mitigate this scenario, nodes propagate their raw and summary information to their parent and children in either a lazy (background) or on-demand fashion.

2.3.3 Mobile Ad Hoc Networks

Data management and aggregation in MANETs [21, 34, 47, 59, 62, 64, 101] is similar to that of DIMS and SPEs; however, typical MANETs require energy-efficient operation and have dynamic network topologies. Further, most use unreliable transport protocols, since reliable transport protocols consume more energy. In the general approach for the more common case of robust data aggregation protocols over unreliable transport protocols, locally known attributes are disseminated periodically to other nodes. When a node receives attribute information, it merges it with its local information such that as the node receives more partial estimates from its peers, its local estimate converges to the actual value of the global aggregate. Figure 2.3 shows a simple example of a such protocol for *average*. Local state is maintained as $sum/count$, and, initially, each node only knows about its local information. Each time a node exchanges data with a peer, it updates its estimate of the global aggregate. In the end, each node's estimate of the average matches the actual average value. Existing protocols vary in supported aggregations (usually a subset of min, max, sum, average and count), attribute dissemination mechanism, and attribute merge operation. Different dissemination and merge techniques lead to different protocol *convergence rates*, the rate at which the total mass estimates at the local nodes converge to the actual total mass.

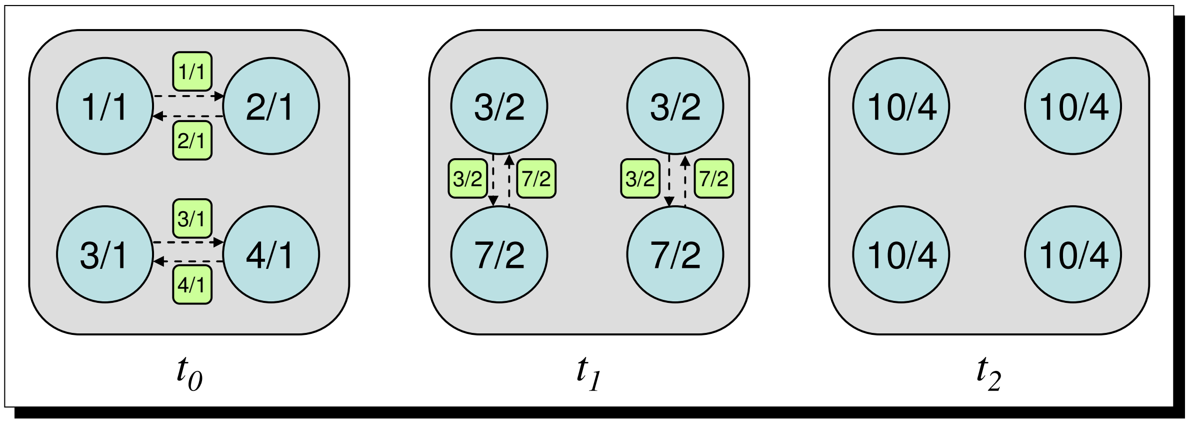


Figure 2.3 Robust MANET aggregation of *average*: Each node maintains an estimate of the average, $sum/count$. At timesteps t_0 and t_1, messages are exchanged to update local estimates of the total average. After two nodes exchange information, they have the same view of the system. At timestep t_2, each node's estimate of the average matches the actual value.

2.3.4 Discussion of Reliable Aggregation

Both Borealis and the system proposed by Hwang et al are generally applicable to all SPE (and, therefore, TBŌN) computations, since they leverage general fault-tolerance approaches based on explicit replication. In contrast, we use specific computational properties to avoid the overhead of explicit replication. For computations that do not exhibit our assumed properties, we might leverage these SPE approaches. Hwang et al also observed that many computations can leverage failure recovery models in which output temporarily diverges from that of the equivalent failure-free computation and that the output of the failed computation is still semantically correct. Our failure recovery model assumes a similar data consistency model.

In general, DIMS and TBŌNs address different communication goals: DIMS support the more general paradigm by aiming to disseminate global information to all nodes in the system. Astrolabe's unstructured gossiping leads to a high amount of replication – eventually, all aggregates will be replicated at all nodes. As a result, Astrolabe is designed for applications with small data sets (hundreds to thousands of bytes) that do not require low communication latencies. In contrast, our model targets applications with with low latency, high bandwidth requirements.

Gupta et al use a semi-structured hierarchical model based on parent/children relationships, but a parent and its children communicate using unstructured gossip. Because gossiping is inherently non-deterministic, gossip-based protocols are resilient to unreliable message delivery. As a result, gossip-based are only suited for applications that can tolerate partial results that estimate global information.

SDIMS addresses some of these issues by using structured, tree-based communication patterns and allowing the user to control the extent of replication. As a result, SDIMS incurs potentially high replication overhead but can accommodate general data aggregation operations. We eschew the overhead of explicit replication by using the inherent redundancies found in broad classes of aggregation operations.

Like TBŌNs, MANETs typically aim to make global information available at a single point, such as a base station. However, these resource-starved environments must employ extremely lightweight protocols based on unreliable message delivery. Many protocols for providing global estimates based on partial data have been proposed, many with good scalability characteristics and convergence rates. However, each protocol is specific to particular aggregation functions, and protocols have been proposed for relatively simple aggregation operations like *sum*, *max*, and *average*. We target a broader set of more complex aggregation operations on possibly voluminous data sets. Further, our approach is not dependent on the specific aggregation operations being applied; we only require that the aggregation operations exhibit certain fairly general properties (Section 3.2.2). We can also leverage reliable transport mechanisms to provide a deterministic, reliable message service.

2.4 Tree Reconstruction

Prior research in tree (re)-construction for (overlay) networks appears in a variety of contexts including MANETs and overlay networks. In this survey, we do not consider algorithms used to initially construct trees; these algorithms could be used to repair failed trees, but at a high and non-scalable cost. Instead, we consider only lightweight approaches to repair disconnected spanning trees. Also, we do not consider tree reconstruction in MANETs, since topology organization in these environments is constrained by the limited communication range of nodes, like wireless sensors. Instead, we study environments that do not have these constraints.

Overcast [46] and the Host Multicast system [107] reconstruct partitioned spanning trees by having orphaned nodes consider their ancestry. In Overcast, an orphan is adopted by its closest surviving ancestor in the tree; nodes periodically measure the network latencies between themself and their siblings and grandparents to re-evaluate their position. The Host Multicast system uses a similar approach but considers network performance when an orphaned node seeks a new parent. These approaches do not consider collisions caused by multiple orphans making the same choices for new parents potentially resulting in imbalanced trees that may perform poorly. Additionally, during failure recovery, the Host Multicast system's performance measurements increase failure recovery latencies.

Another approach for tree reconstruction is to organize orphaned nodes into a single spanning tree rooted at one of the orphans. The root of this tree is then reconnected to the main tree trunk. Yang and Fei [103] proposed a proactive approach in which parent nodes pre-compute the spanning tree that their children will use, should they fail. They used an ancestor list, like the ones used in Overcast and Host Multicast, in case the prescribed new parent is also unavailable. Saia and Trehan [80] proposed a similar approach to Yang and Fei's, except orphaned nodes are organized into a binary tree and the new tree is calculated reactively (post failure). Our results show that such approaches, which only consider a small subset of the network for adoption, can lead to imbalanced trees with large fan-outs, leading to less efficient tree performance. Additionally, the failure time coordination of the reactive approach reduces failure recovery responsiveness.

Banerjee, Bhattacharjee and Kommareddy designed an application-layer multicast protocol that organizes nodes into hierarchical clusters, similar to those of Astrolabe [75] and Gupta et al's data aggregation approach [40]. The node at the graph theoretic center of each cluster acts as that cluster's agent. The hierarchy becomes partitioned when an agent dies, and the new cluster center assumes the agent's role. This approach results in a single node (one of the orphans) adopting all other orphans leading to imbalanced trees. In our computational model, in which waves of data are being aggregated at all levels at all times, imbalanced trees can perform poorly. Additionally, the inter-process communication necessary to identify the new cluster center reduces the responsiveness of failure recovery.

Finally, the Application Level Multicast Infrastructure (ALMI) [71] uses a central *session controller* to organize nodes into a minimum spanning tree. The session controller always dictates the topology based on inter-node performance reports and node departure and arrival events. This centralized approach is not responsive because all orphans must interact with the session controller sequentially; additionally, centralized approaches do not scale to large network sizes. In our recovery model, each orphan makes reconstruction decisions that consider collisions without inter-process coordination.

2.5 Summary of Related Work

The largest distinction between existing failure recovery models and ours is that existing mechanisms employ explicit data replication that can limit their scalability. In contrast, our recovery model leverages inherent information redundancies based on properties of the data aggregation operations to completely avoid explicit data replication.

As in our model, other approaches that are specifically designed for data aggregation operations rely on weak data consistency models. The gossip-based approaches and the failure recovery approaches for MANETs are resilient to unreliable message delivery, but their non-deterministic nature makes them unsuitable for applications that cannot tolerate missing output. Our approach relies on reliable message delivery but provides a deterministic message delivery with no missing output. Unlike protocols based on unstructured gossiping, our model targets applications with low latency, high bandwidth requirements. Additionally, the MANET approaches rely on semantics of aggregation operations that make them suitable for only a limited set of operations.

For tree reconstruction, existing techniques are deficient in at least one of the following areas: they do not consider orphan collisions that may lead to imbalanced trees; performance measurements and other new parent negotiations may limit failure recovery responsiveness; they consider only a small subset of the tree for adopting orphans potentially leading to imbalanced trees, or they use non-scalable centralized mechanisms. We address all these issues by using completely localized algorithms that consider orphan collisions and all the available parents in the tree.

Chapter 3

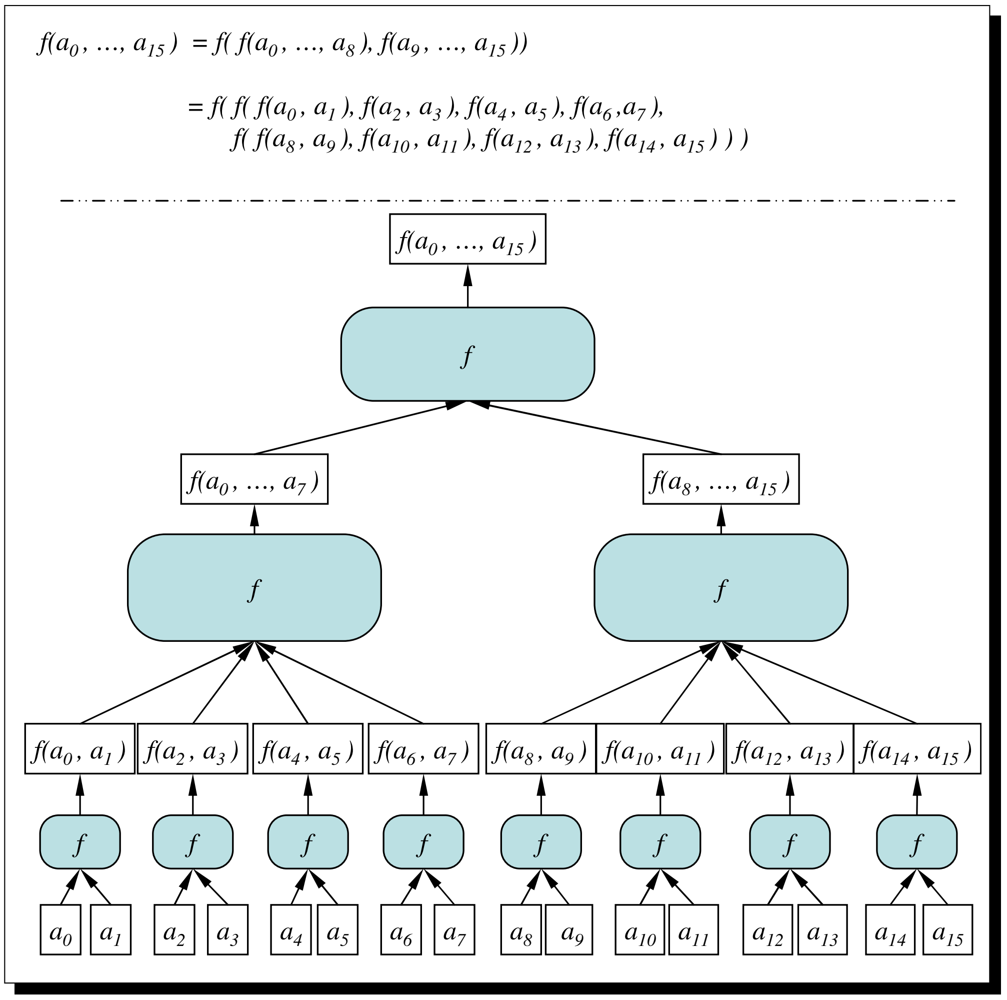
Tree-based Overlay Networks

This dissertation is a study of scalable failure recovery mechanisms for TBŌN-based data aggregation. We now describe the general approach to TBŌN-based data aggregation, and follow it by a formal specification of this computing model. We use this specification in later chapters to prove the correctness of our failure recovery model. We conclude this chapter with a discussion of different types of tools and applications that leverage the TBŌN data aggregation model.

3.1 The TBŌN Data Aggregation Approach

The aggregation model provided by TBŌNs is based on the well-known functional decomposition technique known as *divide and conquer* [11]. As shown in Figure 3.1, the divide and conquer strategy recursively breaks down data aggregation problems into smaller sub-problems. For TBŌN data aggregation, the problem is decomposed by sub-dividing its input set. The TBŌN then maps these smaller sub-problems to different computational resources to compute their solutions efficiently.

This model of computing generally befits functions with the following characteristics:



$f(a_0, \ldots, a_{15}) = f(f(a_0, \ldots, a_8), f(a_9, \ldots, a_{15}))$
$= f(f(f(a_0, a_1), f(a_2, a_3), f(a_4, a_5), f(a_6, a_7)), f(f(a_8, a_9), f(a_{10}, a_{11}), f(a_{12}, a_{13}), f(a_{14}, a_{15})))$
$f(a_0, \ldots, a_{15})$
f
$f(a_0, \ldots, a_7)$
$f(a_8, \ldots, a_{15})$
$f(a_0, a_1)$ $f(a_2, a_3)$ $f(a_4, a_5)$ $f(a_6, a_7)$ $f(a_8, a_9)$ $f(a_{10}, a_{11})$ $f(a_{12}, a_{13})$ $f(a_{14}, a_{15})$
a_0 a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8 a_9 a_{10} a_{11} a_{12} a_{13} a_{14} a_{15}

Figure 3.1 Divide and Conquer: TBŌNs use *divide and conquer* to decompose aggregation functions into smaller more manageable problems.

1. *Associativity*: the output is independent of the grouping of its input elements;

2. *Commutativity*: the output is independent of the ordering of its input elements;

3. *Input/output type equivalence*: the output is of the same type as its inputs. For example, if the inputs are sets of elements, the output is a set of elements;

These characteristics are necessary for decomposed aggregation functions to provide the same functionality as their composite counterparts. Associativity and commutativity allow the input elements to be distributed flexibly amongst the functional components of the decomposed solution. Input/output type equivalence allows the functional sub-components to be composed arbitrarily without considering input/output type agreement issues.

The primary motivation for using TBŌN-based function decomposition is to improve performance. When the sub-components of a decomposition, as in Figure 3.1, are executed on independent processing units, performance generally is improved if the function is a *data reduction*, and its run time is based primarily on its input size. APL [45] introduced data reduction as a programming language concept with its binary operator, "/". The left operand is a function and the right operand is an array, and the operator returns the result of applying the function operand to all elements of the array. For example, $+/1, 2, 3, 4$ will output 10. Data reduction operations generally rely on associativity and commutativity and often summarize large amounts of data into smaller data, such that in a TBŌN, the amount of data propagated to each process does not grow but instead remains relatively constant. The decomposed function's run time performance is based on the tree's fan-out. The run time complexity of the composite version is $O(N)$, where N is the total number of tree processes.

While it may not be immediately obvious, *holistic* functions [39], data aggregation operations that are not data reductions, may also benefit from the TBŌN data aggregation model. For example, *concatenate* is not a data reduction, but we have shown that hierarchical concatenation can yield significant performance improvements, because it is more efficient to transmit and receive a single large data packet than many small ones [76].

3.2 A Specification of the TBŌN Computational Model

Figure 3.2 is a depiction of our TBŌN computational model. For simplicity, many of our illustrations show balanced, binary trees; the general model only requires fully connected trees. The figure shows the application[1] back-ends at the TBŌN leaves producing input data and the application front-end at the root consuming the TBŌN's output. As the continuous stream of inputs produced by the application back-ends propagates toward the front-end, the intermediate TBŌN processes aggregate the dataflow. In this section, we formalize the TBŌN computational model that we presume throughout this dissertation, pointing out how our model differs from the general TBŌN computational model and the implications of these differences.

3.2.1 Data Communication

Collectively, the application 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. An

[1]We use the term application to describe the software system directly leveraging the TBŌN, whether it be a software tool or an actual application.

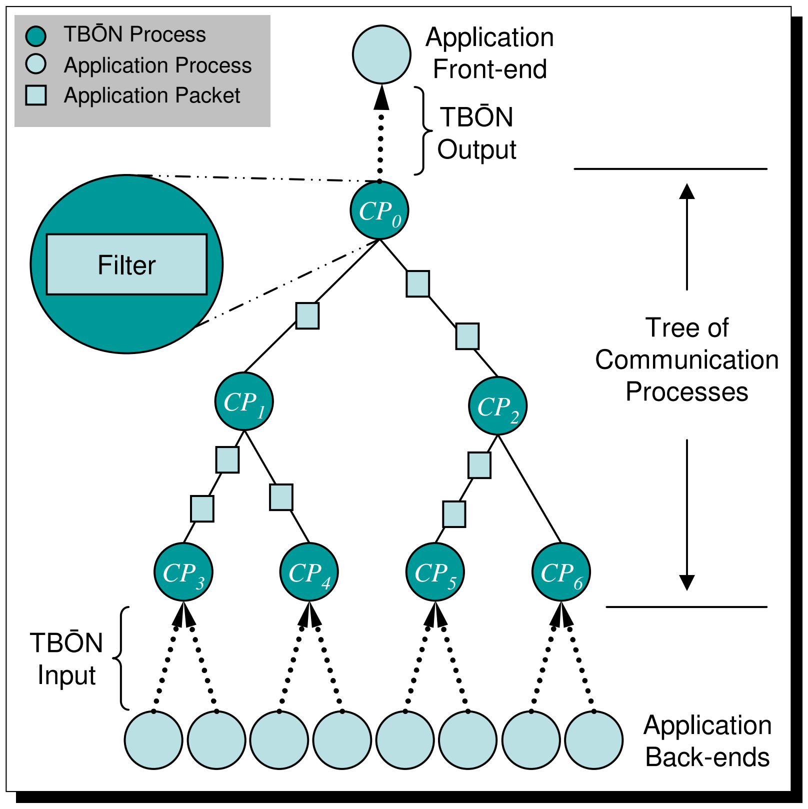
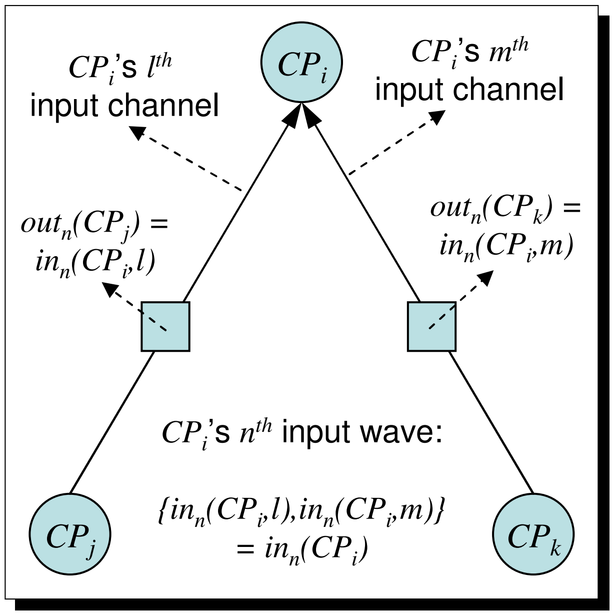


Figure 3.2 The TBŌN Computational Model. Application back-ends continuously stream data into the TBŌN. Communication processes use filters to aggregate this data and propagate aggregation results to the front-end.

application front-end uses *streams* to multicast data to and gather data from groups of back-end processes. A stream specifies the end-points participating in a logical dataflow and distinguishes packets belonging to different dataflows. A stream also specifies the aggregation operation to be applied to packets that flow on that stream: data aggregation operations (described below) can be used for reduction of data from the back-ends to the front-end. Data packets flowing on different streams may be aggregated using different aggregation operations.



CP_i's l^{th} input channel
CP_i
CP_i's m^{th} input channel
$out_n(CP_j) = in_n(CP_i,l)$
$out_n(CP_k) = in_n(CP_i,m)$
CP_i's n^{th} input wave:
$\{in_n(CP_i,l), in_n(CP_i,m)\} = in_n(CP_i)$
CP_j
CP_k

Figure 3.3 TBŌN Input/Output

Parent communication 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. Naturally, a child process' outputs eventually become its parent's inputs:

$$out_n(CP_j) = in_n(CP_i, l) \tag{3.1}$$

where CP_j is the source for CP_i's l^{th} channel. We show this scenario for a parent with two children is shown in Figure 3.3.

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

$$cs_{m,n}(CP_i, j) = [in_{m+1}(CP_i, j), \ \ldots \ , in_n(CP_i, j)] \tag{3.2}$$

$cs(CP_i)$ represents the set of the channel state from all of CP_i's input channels:

$$cs(CP_i) = \bigsqcup_{j=0}^{fanout(CP_i)-1} cs(CP_i, j) \tag{3.3}$$

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

3.2.2 Data Aggregation

TBŌN communication processes use *filters* to aggregate input data packets from their children. We adopt the dataflow model [48] in which 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*; as shown in Figure 3.3, $in_n(CP_i)$ designates CP_i's n^{th} wave of input data:

$$in_n(CP_i) = \{in_n(CP_i, 0), \ in_n(CP_i, 1), \ \ldots \ , \ in_n(CP_i, fanout(CP_i) - 1)\}. \tag{3.4}$$

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 over time. 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 [77], which continuously merges input sub-graphs into a single graph. Each communication process stores as its state the current merged graph, which encapsulates the history of sub graphs filtered by that process. As new sub-graphs arrive, the filter only needs to output incremental changes to its current merged graph (filter state).

A filter's state is initialized to *null*, $fs_0(CP_i) = \varnothing$, and $fs_n(CP_i)$ is CP_i's filter state after it has filtered n waves of data. Using our notation a filter function, f, is defined as:

$$f(in_n(CP_i), fs_n(CP_i)) \rightarrow \{out_n(CP_i), fs_{n+1}(CP_i)\} \tag{3.5}$$

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[2]. A filter instance operates on a specific stream or dataflow; 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.

3.2.2.1 State join

Our join operator, $\sqcup$, merges individual input packets to comprise an input wave:

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

where $fanout(CP_i)$ returns the fanout at CP_i. Also using this join operator, a filter updates its current state by merging it with these input waves:

$$in_n(CP_i) \sqcup fs_n(CP_i) \rightarrow fs_{n+1}(CP_i) \tag{3.6}$$

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

$$fs_n(CP_i) = in_0(CP_i) \sqcup \ldots \sqcup in_{n-1}(CP_i). \tag{3.7}$$

[2]The general TBŌN model allows multiple outputs, but we have not found a practical need for this.

Our model presumes that the join operation has the following properties:

$$\textbf{Associativity}: \quad (a \sqcup b) \sqcup c = a \sqcup (b \sqcup c)$$

$$\textbf{Commutativity}: \quad a \sqcup b = b \sqcup a$$

These properties are admissive of many useful computations[5, 6, 49, 63, 76, 77]. The relevance of associativity and commutativity for TBŌN-based data aggregation has been discussed in Section 3.1. For our failure recovery model, these properties relax the constraints of tree reconstruction mechanism of Chapter 7. Since the computation's correctness does not depend on the grouping and ordering of input data, when failures occur the TBŌN does not have to preserve the original operand order or grouping, and disconnected sub-trees are allowed to reconnect to any branch of the main tree.

3.2.2.2 State difference

Filter functions based on idempotent join operators, for which $\forall x, x \sqcup x = x$, may output either incremental or complete updates. Filter functions based on non-idempotent join operators can output only incremental updates so that they avoid processing the same input data more than once. For efficient run time operation, we favor filter functions outputting the incremental difference between their previous and current states:

$$fs_{n+1}(CP_i) - fs_n(CP_i) = out_n(CP_i) \tag{3.8}$$

However, for failure recovery purposes, we would like the option of sending complete updates, and in this case, non-idempotent operations complicate the recovery mechanisms as detailed in Section 6.5.

Many data aggregation operations, including the majority of the existing MRNet-based data aggregation operations [5, 6, 76, 77], are idempotent. Specific examples 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.

Our failure recovery model depends upon inherent information redundancies amongst the filter state of communication processes and their descendants. As we describe in Section 6.3.1, this inherent redundancy is based upon the equivalence of the filter's input and output – intuitively, the output is a summarized form of the input. The concept of *invertibility*, being able to compute inputs from output and vice versa, is the basis for this input/output equivalence. We discuss two concepts of invertibility, the traditional mathematical invertibility and a more general condition we call *contextual invertibility*

A function f is *invertible* if $f(x_1, x_2) \rightarrow y$ and there exists f^{-1} such that:

$$f^{-1}(y, x_1) \rightarrow x_2, \text{ and } f^{-1}(y, x_2) \rightarrow x_1.$$

In our computational model, this requires that

$$\forall a \text{ and } \forall b,\ a \sqcup b = c,\ c - a = b,\ \text{and } c - b = a.$$

Addition and subtraction, multiplication and division, power and root, and exponential and logarithmic operations are examples of invertible mathematical operations. It can be shown that if two functions f and g are invertible, then their composition, $f \circ g$, is invertible by $g^{-1} \circ f^{-1}$. Therefore, complex functions composed of invertible operations are also invertible. Aggregation operations based on the merging or classification of data structures with summation or magnitude features are also invertible. Consider the example where $\sqcup$ is set union, and $-$ is set difference and set members are a tuple, $\{integer, occurrences\}$:

$\{\{1,1\},\{2,1\},\{3,1\}\} \sqcup \{\{2,1\},\{3,1\},\{4,1\}\} = \{\{1,1\},\{2,2\},\{3,2\},\{4,1\}\}$; and

$\{\{1,1\},\{2,2\},\{3,2\},\{4,1\} - \{\{1,1\},\{2,1\},\{3,1\}\} = \{\{2,1\},\{3,1\},\{4,1\}\}$; and also

$\{\{1,1\},\{2,2\},\{3,2\},\{4,1\} - \{\{2,1\},\{3,1\},\{4,1\}\} = \{\{1,1\},\{2,1\},\{3,1\}\}$.

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 and the set members are scalar elements, for example integers:

$\{1,2,3\} \sqcup \{2,3,4\} = \{1,2,3,4\}$, but

$\{1,2,3,4\} - \{1,2,3\} = \{4\} \neq \{2,3,4\}$; however

$\{1,2,3\} \sqcup \{2,3,4\} = \{1,2,3\} \sqcup \{4\}$.

Therefore, $\{2,3,4\} \equiv \{4\}$ in the context of joining with $\{1,2,3\}$.

Contextually invertible functions are a strict superset of invertible functions: all invertible functions are contextually invertible, but not vice versa.

For idempotent operations, we require only that “$-$” be the contextual inverse of “$\sqcup$”. Contextual invertibility is sufficient because, since processing the same input data multiple times does not affect the computations output, a child process does not need to propagate input data that it previously has propagated to its parent. For non-idempotent operations, “$-$” must be the precise inverse of “$\sqcup$”, since every input data element must be processed exactly once.

3.2.2.3 A TBŌN Data Aggregation Example

In Figure 3.4, we provide an example of the TBŌN data aggregation model using an integer union computation. In this computation, integer input data are propagated through the TBŌN from the leaves to the root. Each process suppresses duplicates values in its input and sends the unique values to its parent. The persistent filter state at each process contains that process′ set of previously filtered integers; the state of each channel is the incident vector of pending incremental updates transmitted from the child of the channel to its parent. The final output at the application front-end is the overall set of unique integers input by the back-ends. In this example, $\sqcup$ is set union, $-$ is set difference.

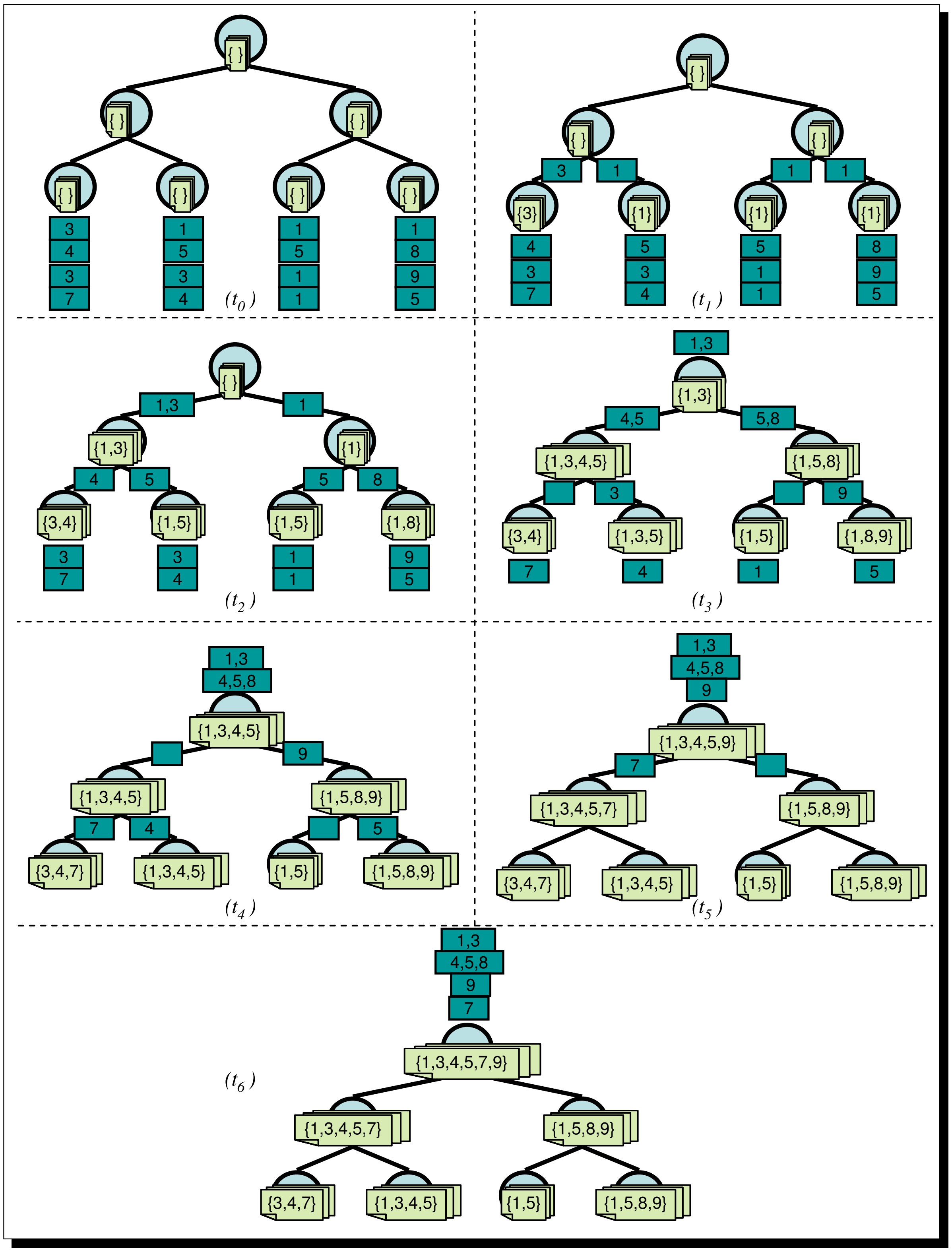


Figure 3.4 TBŌN Integer Union: "⊔" is set union. "-" is set difference. fs is input history.

3.2.2.4 Summary of Notation

We conclude this section with a summary of our TBŌN notation, which will be used throughout the rest of this dissertation.

N	$\Rightarrow$	number of communication processes
CP_i	$\Rightarrow$	i^{th} communication process; $0 \leq i \leq N-1$
$fanout(CP_i)$	$\Rightarrow$	number of CP_i's input channels
$in_n(CP_i, j)$	$\Rightarrow$	n^{th} input wave to CP_i on its j^{th} channel
$in_n(CP_i)$	$\Rightarrow$	$\bigsqcup_{j:0}^{fanout(CP_i)-1} in_n(CP_i, j)$
$out_n(CP_i, j)$	$\Rightarrow$	n^{th} output of CP_i to its parent
$cs_{m,n}(CP_i, j)$	$\Rightarrow$	channel state for CP_i on its j^{th} after CP_i has filtered m waves and the channel's source has sent n packets, $m \leq n$
$cs(CP_i)$	$\Rightarrow$	$\bigsqcup_{j:0}^{fanout(CP_i)-1} cs(CP_i, j)$
$fs_n(CP_i)$	$\Rightarrow$	filter state at CP_i after it has filtered n input waves

Table 3.1 Summary of TBŌN Notation

Chapter 4

MRNet: The Multicast/Reduction Network

MRNet [76] is our prototype of the TBŌN model described in Chapter 3. MRNet has two main components: the first is *libmrnet*, a C++ library that is linked into an application's front-end and back-end processes. The second component is *mrnet_commnode*, the program for the communication processes, which comprise the TBŌN process tree. In this chapter, we describe the features of these components that are relevant to this work.

4.1 MRNet Overview

We use a simple aggregation, *integer maximum*, to direct our discussion of MRNet's components. Figure 4.1 shows the source code for the MRNet front-end, back-end and filter function of this example. The application performs the aggregation on data propagated from the back-ends to the front-end. In line 2, the front-end creates a new Network object, which instantiates the TBŌN process tree. In line 3, the network object is queried for the default *broadcast communicator*, which contains all the back-ends in the network. In line 4, a new stream is bound to the broadcast communicator; the

WAIT_FOR_ALL synchronization filter and the *INT_MAX* transformation filter will be applied to packets sent on this stream.

Once a stream is established, the application end-points can use it for scalable communication. In line 5, the front-end sends the message “go” to the back-ends. In line 11, the back-ends receive this message and, in line 13, respond with a random integer. Finally, in lines 6 and 7, the front-end receives and unpacks the packet that contains the maximum of the integers sent by the back-ends.

Lines 16–21 show the implementation of the integer maximum filter. It calculates the maximum value of the data contained in its input packets and creates a new output packet with that value. This value is then placed in the output packet vector and the filter returns. We now describe the details of this example and the supporting MRNet features.

4.2 MRNet Process Tree Instantiation

To use MRNet for scalable data communication and aggregation between the front-end and back-ends, the application must first instantiate the MRNet process tree. As shown in Figure 4.1, line 2, the front-end instantiates the process tree by creating a *Network* object using the *Network* constructor:

```
Network::Network( const char  * inTopology
                  const char  * inBackEndExe,
                  const char ** inBackEndArgv,
                  ... );
```

The input topology configuration, *inTopology*, dictates how MRNet maps internal and back-end processes to physical hosts as well as the connections between these processes.

```
              /*** MRNet Front-End Code ***/
main() {
  Network *net = new
    Network( topology, backend_exe, backend_args ... );

  Communicator *comm = net->get_BroadcastCommunicator();

  Stream *stream =
    net->new_Stream( comm, INT_MAX, WAIT_FOR_ALL, ... );

  stream->send( PROT_BEGIN, ``%s'', go );
  stream->recv( &tag, &packet );
  packet->unpack( ``%d'', result );
}

              /*** MRNet Back-End Code ***/
main() {
  Network *net = new Network( ... );

  net->recv( &tag, &packet, &stream );

  if( tag == PROT_BEGIN ) {
    stream->send( PROT_INT_DATA, ``%d'', rand_int );
  }
}

               /*** MRNet Filter Code ***/
int_max_filter( vector<Packet> inPackets,
                vector<Packet> outPackets, ... ) {
  for( i=0; i<inPackets.size; i++ )
    result = max( result, inPackets[i].get_int());

  Packet out( PROT_INT_DATA, ``%d'', result );
  outPackets.pushback( out );
}
```

Figure 4.1 Sample MRNet Code

inBackEndExe and *inBackEndArgv* are used to create the back-end processes: the former specifies the executable program for the back-end processes, and the latter is the list of command line arguments to be passed to the back-ends.

MRNet supports two network instantiation modes: in the first mode, MRNet creates both the internal communication and back-end processes. This process is illustrated in Figure 4.2. First, the front-end uses a remote shell mechanism, like *rsh* or *ssh*, to create its children processes for the first level of the communication tree. Each newly created child process establishes a connection back to its parent process and receives the portion of the topology configuration relevant to that child. Each child then uses this information to instantiate its immediate children. This procedure is repeated until the entire tree of communication and application processes is created.

In the second mode, the internal communication processes are instantiated just as in the first mode, but the back-end processes are created by some third party mechanism. Even though the back-end processes are not created by MRNet, the topology configuration specifies where the back-ends are located in the topology. For this instantiation mode, MRNet provides two additional API routines:

```
struct BackEndInfo {
  Rank backend_rank,
  string parent_hostname,
  Port parent_port
};

void get_BackEndInfo( vector<BackEndInfo> &outBackEndInfo );

int connect_BackEnds( void );
```

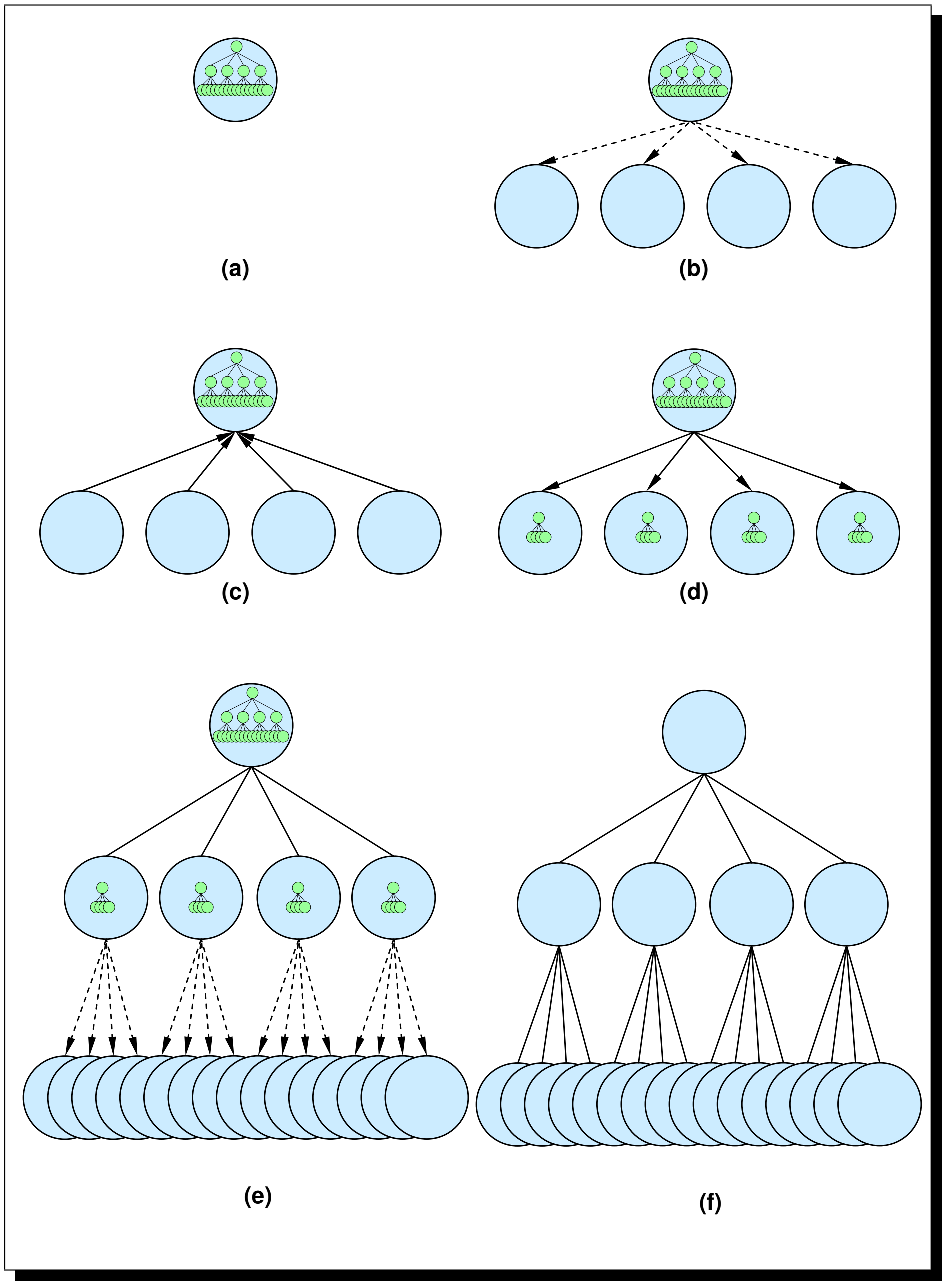


Figure 4.2 MRNet Instantiation: The circles are the TBŌN processes; the tree structure inside a circle represents that process' relevant portion of the topology configuration. (a)The front-end starts with the specified topology configuration, which is used to create the process tree. (b) Based on the topology configuration, MRNet creates the front-end's children. (c) The newly created children connect back to their parent and (d) receive their portion of the topology configuration. Steps b-d are repeated as necessary until the entire process tree is instantiated.

The front-end uses *get_BackEndInfo* to query where each back-end must connect into the network. This method returns a vector of *BackEndInfo* structs, one per back-end process; each struct contains the hostname and port of the parent for the specified back-end rank – each MRNet process has a unique rank identifier. This information is passed to the back-end processes and dictates to which parent process they must connect. Before application back-end processes can join the TBŌN, the front-end must invoke *connect_Backends* to put each parent communication process into a *listening* mode where it blocks until **all** its back-end children have connected.

4.3 MRNet Input/Output

Once the MRNet process tree is established, the application can use it for data transfer. This input/output is done using MRNet *streams*, logical communication channels between front-end and back-end processes. In Figure 4.1, line 4, the *new_Stream* method is used to create a stream and bind it to *inComm*, a *communicator*, which (as in MPI [61]) specifies the end-points participating in that stream's dataflow. This method also specifies the filters, *inTransFilter* and *inSyncFilter* (discussed below), used to aggregate data that will flow on the stream:

```
Stream * Network::new_Stream( Communicator *inComm,
                              int inTransFilter,
                              int inSyncFilter,
                              ... );
```

MRNet supports multiple, simultaneous streams of communication, even among the same end-points, within an application instance. So, for example, an application can execute different data aggregations on different streams of data from the same end-points.

As in Figure 4.1, lines 5–7, 11 and 13, application end-points communicate by executing MRNet *send* and *recv* operations on stream objects:

```
int Stream::send( int inTag,
                  const char * inFormatStr,
                  ... );

int Stream::recv( int *outTag,
                  Packet &outPacket,
                  ... );

int Packet::unpack( const char * inFormatStr,
                    ... );
```

The *send* method encapsulates application data into an MRNet *packet*. Applications use the *inTag* and *outTag* parameters to identify a packet's content. *inFormatStr* is a format string, similar to that used by C formatted output routine *printf*, to specify a packet's data type. In our example, Figure 4.1 lines 5, 7, 13, and 19, the format string "%s" describes a packet that contains a null-terminated string, and "%d" describes a packet that contains an integer. MRNet uses this packet type information to properly serialize and de-serialize application data in part so that data aggregation operations can be applied properly to packet flows. The *unpack* method is analogous to the C *scanf* routine and retrieves application data from packets returned by *recv*.

4.4 MRNet Filters

MRNet uses *filters* to aggregate data packets from its children into one or more output packets. As shown in Figure 4.1, lines 16–21, a filter inputs a vector of packets and outputs a vector of packets:

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

As mentioned above, a filter instance is bound to a stream when it is created, thus specifying the aggregation operation to apply to packets flowing on that stream. The filter function also takes a reference to a generic pointer for filter state, *inoutFilterState*. MRNet maintains a unique filter state reference for each filter instance. On first invocation, a filter routine may allocate data structures for its local storage and assign a pointer to its data to the supplied filter state reference. This reference is then made available to the filter instance each time it executes.

MRNet distinguishes between transformation filters and synchronization filters. Transformation filters aggregate data from multiple packets into one or more output packets. Typically, transformation filters input a set of packets, one from each child, and output a single packet.

Synchronization filters provide a mechanism to deal with the asynchronous packet arrival from child processes by organizing data packets into synchronized waves. MRNet supports multiple synchronization primitives and allows users to define their own. In our running example, Figure 4.1 line 4 (and throughout this dissertation), we use the most common synchronization mode, *Wait For All*, in which a wave is comprised of a single packet from every child process. Other synchronization modes are *Don't Wait*, in which packets are propagated to the transformation filter as soon as they arrive, and *Time Out*, in which packets are propagated when a complete wave is available or when a configurable time-out expires.

MRNet allows application developers to implement and add new filters using the *load_FilterFunction()* method:

```
int load_FilterFunction( const char * inSharedObject,
                         const char * inFilterFunction );
```

This method dynamically loads the filter routine, *inFilterFunction*, into the TB$\bar{O}$N processes from the shared object file, *inSharedObject*, using operating system services for managing shared objects (for example, *dlopen* and *dlsym* on UNIX systems). The method returns a integer filter identifier that can be used by the *new_Stream* method to bind the named filter to a new stream.

Chapter 5

Large Scale Application Debugging

We demonstrate the scalability of MRNet and the TBŌN model using the MRNet-based stack trace analysis tool, STAT. As part of a collaboration with researchers from the Lawrence Livermore National Laboratory, we designed and developed STAT to address the debugging and analysis of large scale applications. STAT uses process stack traces to assemble application profiles represented by call graph prefix trees and to identify *process equivalence classes*, sets of processes exhibiting similar behavior. STAT can analyze traces from an application running on all 212,992 processors of the IBM BG/L in less than one second. To the best of our knowledge, STAT is the first such tool to run at scale on the world's largest supercomputer.

5.1 Challenges of Performance and Debugging Tools

Even at relatively modest scales, most current performance and debugging tools perform inefficiently, if at all. For example, TotalView [56], a widely used debugger for HPC environments, takes more than two minutes to collect and merge stack traces from a 4,096 process application on BG/L (4,096 is approximately 2% of BG/L). Developing scalable diagnosis tools presents several challenges [76]:

- *Overwhelming channels of control*: In most parallel tools, a front-end process controls the interactions between back-end tool daemon processes and the debugged application's processes. At large process counts, the front-end can spend unacceptably long times managing the connections to the back-end daemons.
- *Large data volumes*: As the number of debugged processes increases, the volume of data becomes prohibitively expensive to gather.
- *Excessive data analysis overhead*: Even if the debug data can be gathered in an acceptable time, the time to process and to present it becomes excessive, often causing users to resort to targeted print statements.
- *Scalable result presentation*: Debugging and analysis results from hundreds or thousands of processes can overwhelm tool users and prevent quick anomaly detection. We need presentation paradigms that effectively consolidate results from many processes into compact, easy-to-navigate representations.

STAT addresses these challenges by using MRNet to manage the scalable collection, analysis and visualization of stack traces to profile large scale application behavior. This approach was motivated by real application debugging experiences at the Lawrence Livermore National Laboratory from which we observed that:

1. Many program errors only show up beyond certain scales;

2. Program errors may be non-deterministic and difficult to reproduce;
3. Stack traces provide useful insight into an application's behavior;
4. Unexpected behavior often has a temporal aspect – the behavior is erroneous not because it occurs but because it persists; and
5. Processes of parallel computations, even ones that have experienced errors, often can be grouped into a few subsets of processes with similar run time behavior.

Based on these observations, we designed a lightweight mechanism for sampling stack traces over time from large scale application instances and identifying process equivalence classes. Specifically, STAT uses a call graph prefix tree to distinguish process equivalence classes and guide the diagnosis process by allowing the user to focus on single representatives of each behavior class. As shown in Figure 5.1, the nodes of a call graph prefix tree are function names, and the path from the root to any node represents the call path to that function's invocation. Our lightweight, hierarchical diagnosis approach quickly reduces the exploration space from thousands or even millions of processes to a handful of behavior classes (and class representatives). Once the problem space is reduced, the user can perform root cause analysis with a full-featured debugger, since now it is only necessary that this debugger attach to a small subset of the processes.

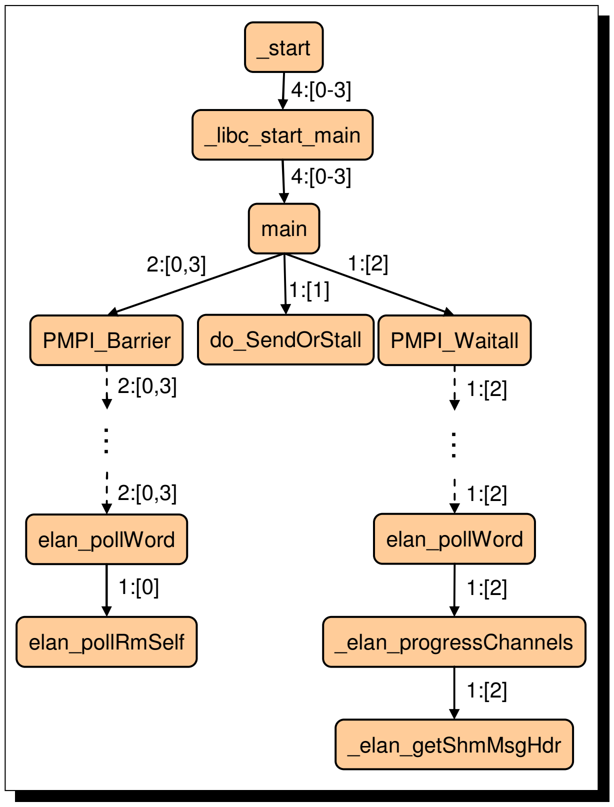


Figure 5.1 Call Graph Prefix Tree:

5.2 Scalable Stack Trace Analysis

As illustrated in Figure 5.2, a stack trace depicts the caller/callee relationships of the functions being executed by a process at the time the stack trace was sampled. Such singleton stack traces are supported by most if not all debuggers, often using a textual representation. However, singleton traces do not allow effective evaluation of large applications: an application with a thousand processes would generate a thousand stack traces – beyond the threshold of easy comprehension.

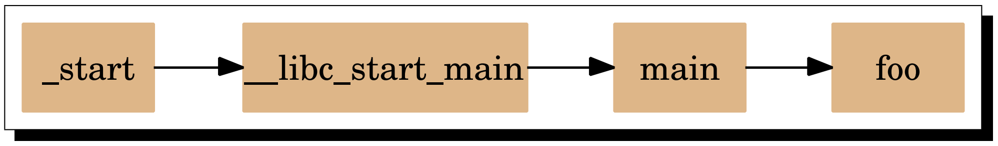


Figure 5.2 A stack trace showing caller/callee relationships of executing functions.

To address the deficiency of singleton traces, tools like Prism [95] and TotalView [56] support what we call a 2D-Space analysis, merging a single stack trace from each application process into a call graph prefix tree. Generally, there is significant overlap amongst the individual stack traces such that the traces from many processes compress into a relatively small prefix tree. Both Prism and Totalview use a non-scalable, single level process hierarchy with the tool front-end directly connected to the back-end processes to collect these traces.

While 2D-Space call graph prefix trees are well-suited for analyzing static scenarios like examining core dumps, they do not include the temporal information necessary to help answer questions about application progress, deadlock/livelock conditions, or performance. To address these issues, we introduced a temporal component into our 3D-Space/Time stack trace analyses: we merge and analyze sequences of stack traces from the target application's processes collected over a sampling interval. The resulting call graph prefix tree then depicts a global profile of the application's behavior for the sampling period.

In Figure 5.3, we show such a profile from a simple MPI program run with 16 processes. In this program, process ranks are organized into a virtual ring within which each process performs an asynchronous receive from its predecessor in the ring followed by an asynchronous send to its successor. Each process then blocks for these I/O requests to complete (via `MPI_Waitall`). A whole program synchronization point (`MPI_Barrier`) follows the ring communication. We inserted a bug into the program that permanently blocks one task before it completes its send operation. In Figure 5.3, we see how STAT distinguishes process equivalence classes by giving each class its own color. After the top-level equivalence class, there are three behavior classes: the first contains 14 properly functioning tasks blocked at the `MPI_Barrier` call at the end of the program. The second class contains the single erroneous process that has stalled in the `do_SendOrStall` routine. The last class contains the process succeeding the stalled one in the virtual ring; this process is blocked waiting for a message from it predecessor, which it will never receive due to the inserted bug. This scenario is representative of the class of bugs in which erroneous tasks perturb the behavior of some, but not all, well-behaved ones.

5.3 STAT Design and Implementation

STAT is comprised of three main components: the tool front-end, the tool daemons, and the stack trace analysis routine. The front-end controls the collection of stack trace samples by the tool daemons, and the collected traces are processed by our stack trace analysis routine. The front-end renders the result, a single call graph prefix tree. The STAT front-end and back-ends communicate via an MRNet process tree, and MRNet filters implement the stack trace analysis algorithm, which aggregates stack trace input

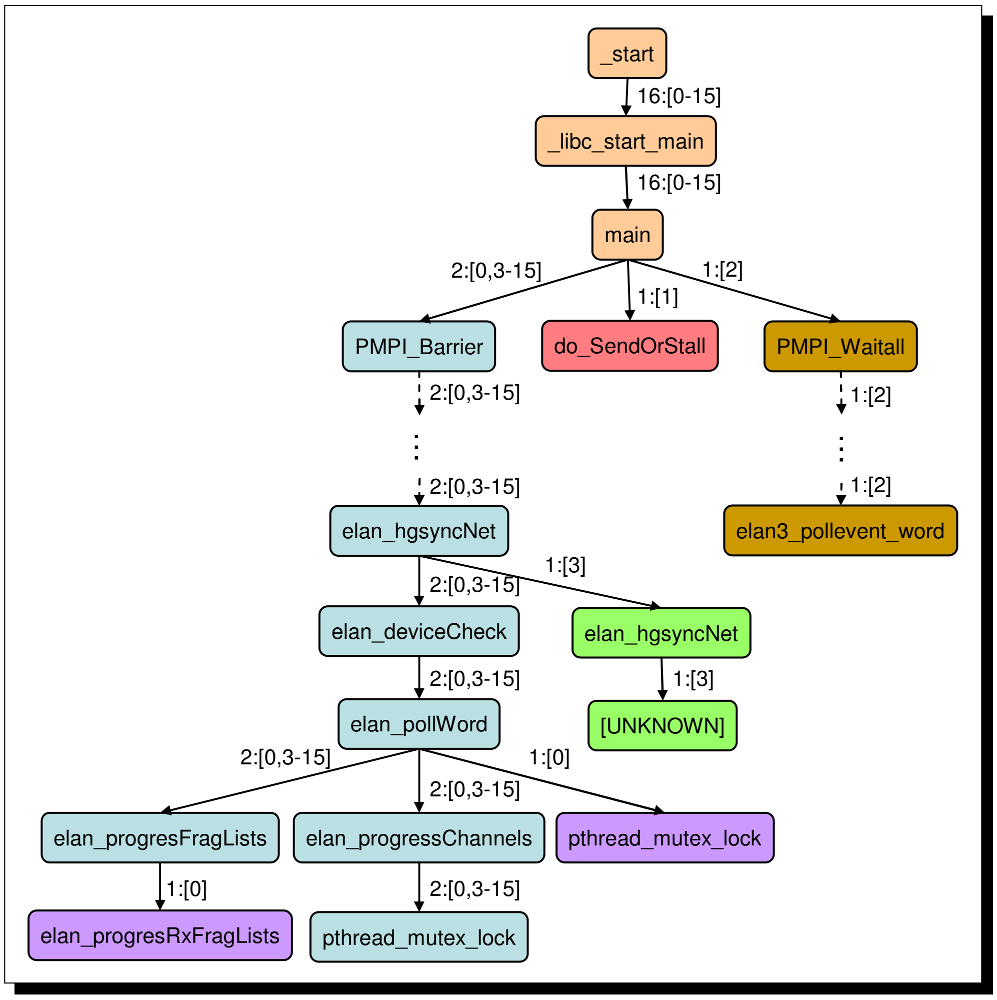


Figure 5.3 A 3D-Trace/Space/Time call graph prefix tree showing a global profile and distinguishing unique behaviors as process equivalence classes.

from children nodes. Back-ends merge locally collected samples before propagating them to their parent processes.

The STAT front-end first instantiates the MRNet tree and tool daemon back-ends. The front-end controls the number of samples and sampling rate of stack traces at the back-ends. Lastly, the front-end receives the single call graph prefix tree that results from merging the individual traces and color-codes the process equivalence classes.

Under the control of the STAT front-end, each STAT back-end attaches to its local application processes, samples process stack traces, merges locally collected samples (using the core function described below) and propagates the results of the local merge up the tree. The back-ends use the Dyninst library [19] to sample stack traces from application processes.

The MRNet processes use the STAT filter to merge input packets from their children. Each packet contains a call graph prefix tree, and the filter's core function merges multiple input call graph prefix trees into a single call graph prefix tree that is then propagated toward the root of the TBŌN to yield a single global tree at the front-end.

5.4 STAT Performance Evaluation

Our first experiments to evaluate STAT's performance and scalability were executed on two high performance clusters at the Lawrence Livermore National Laboratory, Thunder and Atlas. Their specifications can be found in Table 5.1.

For our evaluation, we debugged the MPI message ring program described in Section 5.2 at various scales. The application was run on a node allocation with one MPI task per processor. For debugging, STAT daemons must be collocated with the application processes: we placed one tool daemon process on each node of the application's allocation; that daemon debugged all collocated application processes. Front-end and internal nodes were placed on a separate set of nodes, also with one task per processor.

	Thunder	Atlas
Architecture	Intel Itanium2 (1.4 GHz)	AMD Opteron (2.4 GHz)
OS	Linux (CHAOS 4.0)	Linux (CHAOS 4.0)
Node Count	1024	1152
CPUs/Node	4	8
Memory/Node	8 GB	16 GB
Interconnect	Quadrics QsNet^{II}	InfiniBand

Table 5.1 HPC Clusters used for STAT Performance Evaluation

We evaluated STAT's performance by measuring the time it took to gather and merge the local stack trace samples collected at the STAT daemons into the global prefix tree at the front-end. We omitted the local stack trace sampling period, which is determined by the time to sample an individual stack trace and the number of samples collected and sampling interval as chosen by the user.

We compared the performance of 1-deep trees, the standard tool organization in which the front-end is directly connected to the tool-daemons, to trees with one or two intermediate levels of internal nodes. As scale increases, increasing the tree's depth allowed it to maintain scalable performance. Our results show that for the tested scales, 2-deep trees, trees with a depth of two, were sufficient. All experiments used balanced topologies with internal processes at the same depth having an equal number of children, and we scale these experiments to approximately 4000 thousand application processes. The results are shown in Figures 5.4 and 5.5, respectively. In both cases, as the size of the debugged application increases, the latency of the 1-deep tree grew rapidly with the number of processes being debugged, while latencies in the 2-deep and 3-deep trees increased slowly due to the controlled fan-out.

To evaluate STAT in a large scale environment, we ported STAT to BG/L [52]. During this process, we encountered several scalability issues including:

- *Startup Costs*: Originally, STAT used MRNet's rsh-based instantiation mode, which does not scale well. We modified STAT to leverage native system resource managers for efficiency and portability.

- *File System Issues*: STAT daemons need to access the target application's binary to resolve function names in the sampled stack traces. Standard distributed file systems did not provide scalable mechanisms for concurrent file access by many processes, so we devised a scheme to relocate an executable and its shared libraries from shared file systems to the local nodes' ramdisk for efficient access.

- *Scalable Data Structures*: Originally, STAT used a fix-sized bit vector to represent process equivalence classes. This approach did not allow STAT to run at extremely large scales. We redesigned STAT to use hierarchical, variable-sized vectors that are only as large as necessary.

The results from running the newly designed STAT on BG/L are presented in Figure 5.6. We ran the STAT front-end and internal processes on BG/L's login nodes, and the STAT back-ends ran on BG/L's I/O nodes. Each I/O node manages 64 compute nodes,

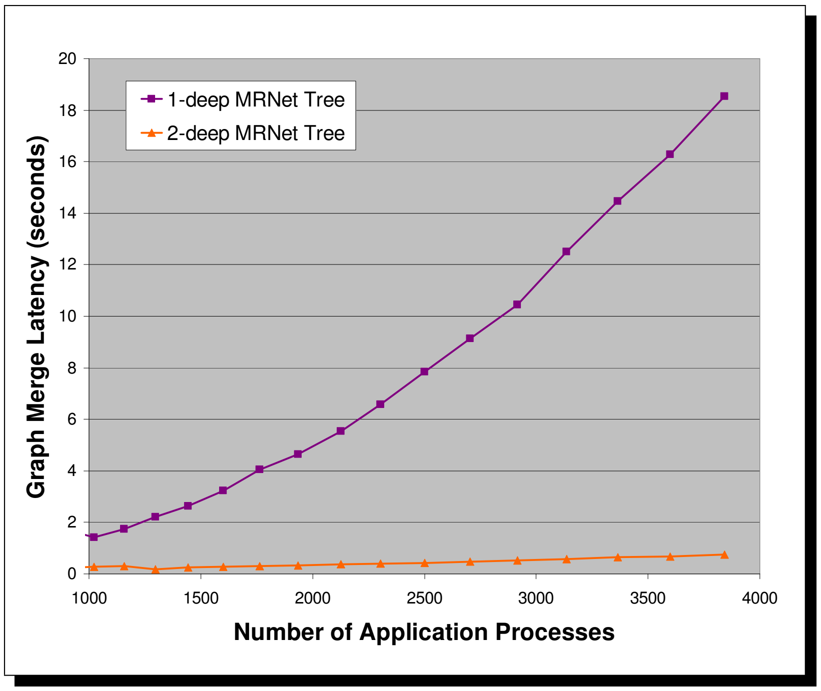


Figure 5.4 STAT Performance on Thunder

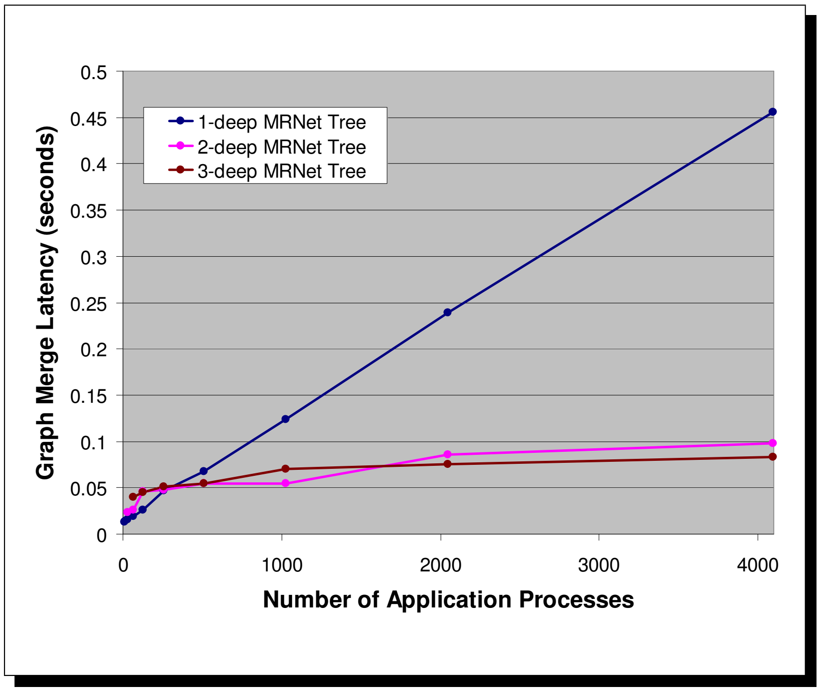


Figure 5.5 STAT Performance on Atlas.

and debugging services on the I/O nodes allow processes to debug other processes running on BG/L's compute nodes managed by that I/O node. For these results, we ran two application processes on each compute node. As with the previous Thunder and Atlas results, the 2-deep and 3-deep trees provided a scalable solution compared to 1-deep process organizations. These results demonstrate that careful design and use of the TBŌN computational model can yield scalable software for the largest of current systems. In this case, STAT debugs an application with 212,992 application processes interactively – with sub-second latencies.

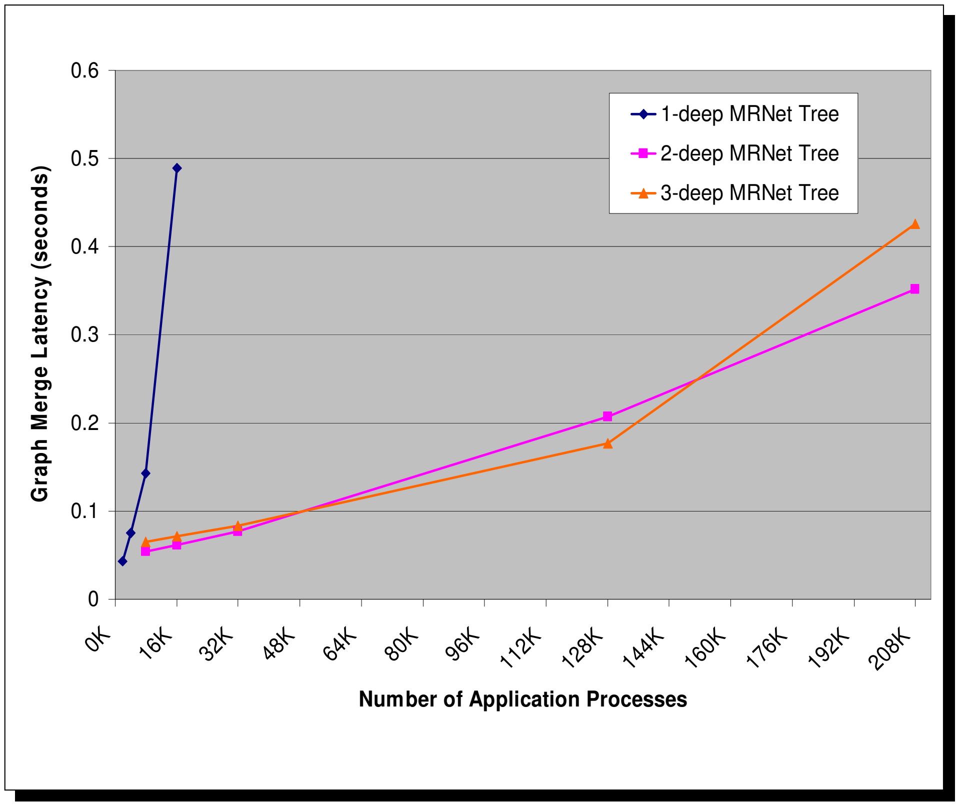


Figure 5.6 STAT Performance on BG/L.

Chapter 6

A Scalable TBŌN Failure Recovery Model

In this chapter, we present *state compensation*, our scalable TBŌN failure recovery model. First, we describe our failure and data consistency models and the TBŌN properties upon which state compensation depends. State compensation uses *space redundancy* [37], the replication of the results of prior computations. However, two key characteristics make state compensation scalable: first, the space redundancy space is inherent to the TBŌN data aggregation operation itself, so the failure recovery technique does not consume any additional time or computational resources during normal operation. Second, for state compensation, lost information is re-transmitted to and filtered by the ascendants of failed processes. However, for each failed TBŌN channel, a single aggregate packet compensates for all the lost packets. In general, this single packet can be filtered more quickly than the original packets that constitute the aggregate.

We have developed two state compensation mechanisms, *state composition* and *state decomposition*. State composition, our primary state compensation mechanism, is lightweight and guarantees at least once input data processing. As such, state composition is suitable for idempotent data aggregation operations, which cover the majority of existing MRNet-based usages.

State decomposition addresses non-idempotent data aggregation operations. The failure recovery mechanisms necessary to guarantee exactly once input data processing involve two coordination phases that may cause temporary throughput delays in the TBŌN application's performance. While we study the performance of state composition in Chapter 8, we do not present a quantitative study of state decomposition's performance or its application perturbation. Nonetheless, we think that state decomposition has some attractive properties that may render it useful despite its coordination complexities: first, like state composition, state decomposition does not require any special fault tolerance mechanisms or message tracking during normal operation. Second, only $O(log(N))$ processes, where N is the number of application back-end processes, participate in the coordination phases, so the approach still may be responsive and scalable.

6.1 Failure Model

Generally, our recovery model tolerates any software or hardware failures that cause TBŌN processes to halt. We do not require failed processes to be replaced before the system can return to normal operation: we redistribute services among the surviving TBŌN processes. The TBŌN's performance should degrade gracefully as processes fail, and failed hosts that are repaired may be re-integrated into the TBŌN. Detail of our failure model specification follow.

We assume the *fail-stop* (also called *fail-silent* and *crash-stop*) failure model in which detectable failures occur due to faults that cause processes to cease producing new output. This failure model does not address *omission failures*, in which processes may fail to produce some intermediate output, or *Byzantine failures*, in which processes may fail arbitrarily. We presume that inter-process communication latency is bounded such that we can distinguish between a slow process and a failed one.

Our system is *n-fault-tolerant*, where *n* is the number of TBŌN processes. An n-fault-tolerant system can tolerate the failure of *n* components. In other words, any TBŌN process (root, leaf or internal) may fail. Should all TBŌN processes fail, the system degrades to the degenerate case with the application's front-end directly connected to all its back-ends. Further, TBŌN process failures may occur at any time, even during the recovery phase of a previous failure. In Section 6.6, we discuss approaches to deal with the failures of application processes, which reside outside the TBŌN.

Hardware failures may cause process failures or the appearance of process failures; for example, hosts or network inter-connection devices may fail. A host failure results in the failures of the TBŌN processes located on that host and are treated accordingly. Network device failures that cause a permanent network partition are treated as failures of the processes partitioned from the TBŌN's root process.

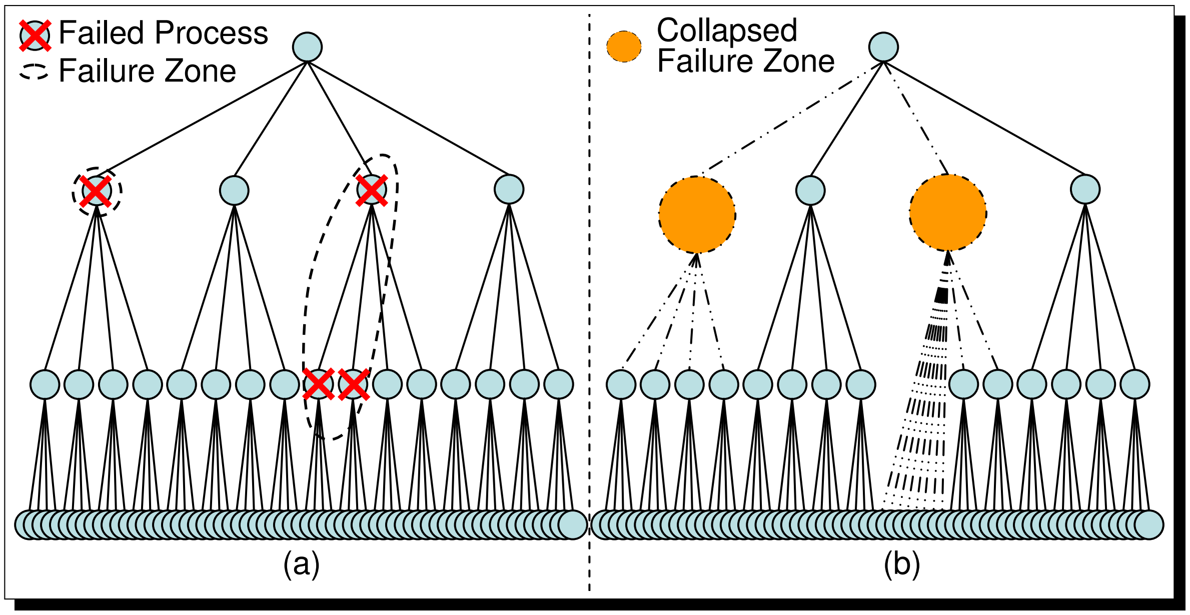


Figure 6.1 TBŌN Failure Zones. (a) Regions of contiguous failures are called *failure zones*. (b) We effectively collapse failure zones into a single process view to tolerate (time) overlapping failures.

We address *overlapping failures*, failures that overlap in time by collapsing failed processes into *failure zones*, regions of contiguous failed processes, as shown in Figure 6.1. Effectively, we treat each failure zone as if it were a single process parenting all the children of the failed processes. That is, that single node possesses the union of the filter state of the failed processes, and the single node's incoming and outgoing channels are those of the failed processes as well. This process is detailed in Sections 6.4.3 and 6.5.4.

6.2 Data Consistency Model

Our recovery model provides weak data consistency, called *convergent output recovery* [43], in which failures may cause intermediate TBŌN output data to diverge from the output produced by the equivalent computation with no failures. Eventually, the post-failure TBŌN output converges back to match the output of its non-failed equivalent.

In convergent output recovery, the output of a computation that has experienced a failure is semantically equivalent, but not necessarily equal to, the output of the same computation without any failures. Our failure recovery model leverages the associativity and commutativity and sometimes the idempotence of TBŌN data aggregation operations to enable lightweight failure recovery mechanisms. Relying on these properties, our failure 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 result in the output divergence. Eventually, the output stream converges back to that of the non-failed computation after all input data affected by the failures are propagated to the root of the TBŌN and it starts to process later input data not affected by the failures.

Figure 6.2 shows output from an *integer maximum* data aggregation and depicts how failures may cause TBŌN output to diverge temporarily. The failure, which occurs at time t_0 causes a divergence in the output stream at times t_1 and t_2; however, at time t_3, the output re-converges to that of the non-failed execution. Convergent recovery preserves all output information and produces no extraneous output information. In Section 6.4, we present an example that demonstrates convergent output recovery.

Output Stream with No Failures:	7	11	27	35	35
Output Stream with Failure at t_0:	7	8	15	35	35
	t_0	t_1	t_2	t_3	Overall Maximum

Figure 6.2 Convergent Recovery for *integer maximum* example. A failure at time t_0 causes a divergence in the output stream at times t_1 and t_2; however, at time t_3, the output re-converges to that of the non-failed execution.

6.3 The Three Fundamental TBŌN Properties

Our goal is to develop scalable state recovery mechanisms for TBŌNs 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 Chapter 3 and, as shown in Figure 6.3, is motivated primarily by three properties (the shaded boxes):

1. *The Inherent Redundancy Lemma*: inherent information redundancies exist within the computational structure of *stateful* TBŌN-based data aggregation operations. Stateful data aggregation operations maintain state that persists across invocations of the operations. Intuitively, as data is propagated from the leaves of the tree toward the root, this aggregation state, which generally encapsulates the history of previously processed input data, 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,

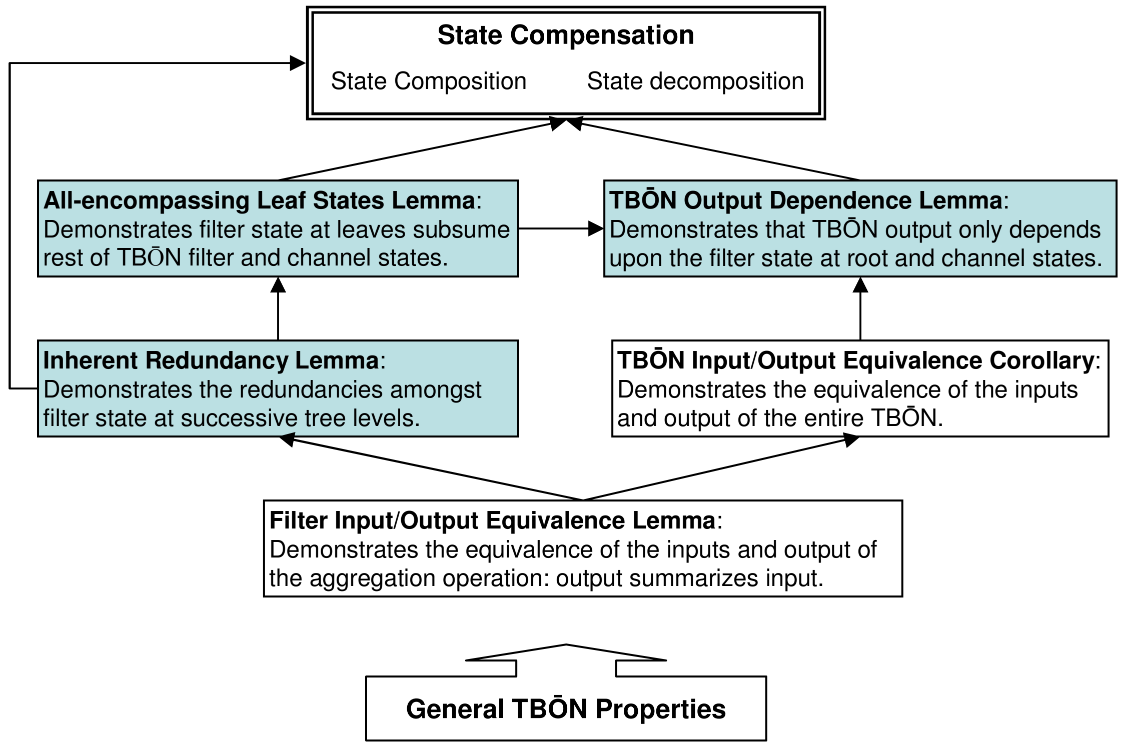

State Compensation
State Composition
State decomposition
All-encompassing Leaf States Lemma: Demonstrates filter state at leaves subsume rest of TBŌN filter and channel states.
TBŌN Output Dependence Lemma: Demonstrates that TBŌN output only depends upon the filter state at root and channel states.
Inherent Redundancy Lemma: Demonstrates the redundancies amongst filter state at successive tree levels.
TBŌN Input/Output Equivalence Corollary: Demonstrates the equivalence of the inputs and output of the entire TBŌN.
Filter Input/Output Equivalence Lemma: Demonstrates the equivalence of the inputs and output of the aggregation operation: output summarizes input.
General TBŌN Properties

Figure 6.3 State compensation leverages three fundamental TBŌN properties demonstrated by the Inherent Redundancy, All-encompassing Leaf States, and TBŌN Output Dependence Lemmas – all derived from the general TBŌN characteristics described in Chapter 3. The arrows in the diagram depict "leads to" property relationships.

in-flight data triggers the execution of data aggregation operations, the output of which depends upon the input data and the current state of the filtering process.

3. *The All-encompassing Leaf States Lemma*: the states at a TBŌN's leaf processes contain all the state 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 the above lemmas, we 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.

6.3.1 Inherent Redundancy

As shown in Figure 6.3, 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 6.1 (Aggregation Input/Output Equivalence) The output of a TBŌN data aggregation operation is equivalent to its given input.

Proof of the Aggregation Input/Output Equivalence Lemma

(Eqn. 3.6: filter state joined with inputs produces the updated filter state.)

$$in_n(CP_p) \sqcup fs_n(CP_p) \quad = \quad fs_{n+1}(CP_p)$$

('$-$' is the inverse of '$\sqcup$'):

$$in_n(CP_p) \quad \equiv \quad fs_{n+1}(CP_p) - fs_n(CP_p)$$

(Eqn. 3.8: fs' - fs = output):

$$\equiv \quad out_n(CP_p)$$

■

For state composition, it is sufficient that "$-$" be the contextual inverse of "$\sqcup$". However, for state decomposition to compute precisely the state information that has been lost due to failures, state decomposition requires the stronger condition that "$-$" be the precise inverse of "$\sqcup$".

Having already discussed the intuition behind the inherent TBŌN information redundancies, we now present its proof:

Lemma 6.2 (Inherent Redundancy) For any TBŌN parent process, the state that results from joining its filter state with its pending channel state is equal to the state that results from joining 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 shown in Figure 6.4, CP_i has two children[1], 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; $m \leq n, m \leq p$.

Proof of the Inherent Redundancy Lemma

(Eqn. 3.7: filter state = join of input history):

$$\begin{aligned} fs_m(CP_i) \sqcup cs_{m,n}(CP_i,0) \sqcup cs_{m,p}(CP_i,1) &= in_0(CP_i) \sqcup \ldots \sqcup in_{m-1}(CP_i) \sqcup \\ &\quad cs_{m,n}(CP_i,0) \sqcup cs_{m,p}(CP_i,1) \end{aligned}$$

(Eqn. 3.4: input = join of children's output):

$$\begin{aligned} &= (out_0(CP_j) \sqcup out_0(CP_k)) \sqcup \ldots \sqcup \\ &\quad (out_{m-1}(CP_j) \sqcup out_{m-1}(CP_k)) \sqcup \\ &\quad cs_{m,n}(CP_i,0) \sqcup cs_{m,p}(CP_i,1) \end{aligned}$$

(Eqns. 3.2 & 3.4: channel state = pending channel source output):

$$\begin{aligned} &= (out_0(CP_j) \sqcup out_0(CP_k)) \sqcup \ldots \sqcup \\ &\quad (out_{m-1}(CP_j) \sqcup out_{m-1}(CP_k)) \sqcup \\ &\quad out_m(CP_j) \sqcup \ldots \sqcup (out_n(CP_j) \sqcup \\ &\quad out_m(CP_k) \sqcup \ldots \sqcup (out_p(CP_k) \end{aligned}$$

[1]We demonstrate our proofs using binary trees. These proofs have straightforward extensions to trees with arbitrary fan-outs.

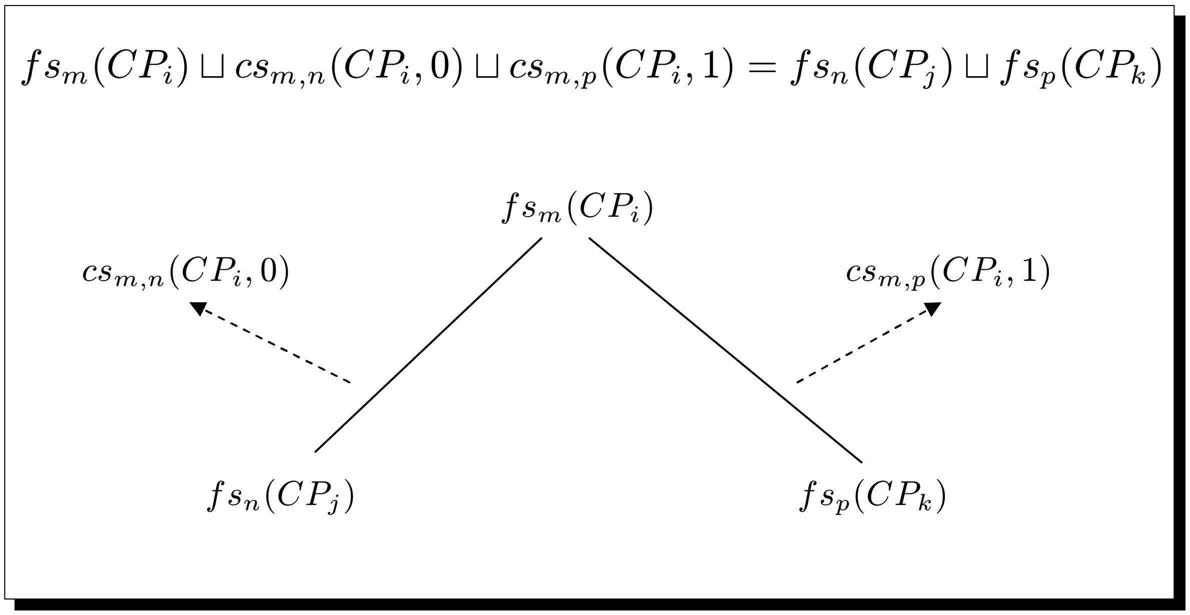

$fs_m(CP_i) \sqcup cs_{m,n}(CP_i,0) \sqcup cs_{m,p}(CP_i,1) = fs_n(CP_j) \sqcup fs_p(CP_k)$
$fs_m(CP_i)$
$cs_{m,n}(CP_i,0)$
$cs_{m,p}(CP_i,1)$
$fs_n(CP_j)$
$fs_p(CP_k)$

Figure 6.4 Inherent TBŌN Information Redundancy: the join of the state at a parent process and its pending channel state equals the join of its children's states. (Only the TBŌN states are shown for simplicity.)

(Commuting the operands):

$$fs_m(CP_i) \sqcup cs_{m,n}(CP_i,0) \sqcup cs_{m,p}(CP_i,1) = out_0(CP_j) \sqcup \ldots \sqcup out_n(CP_j) \sqcup out_0(CP_k) \sqcup \ldots \sqcup out_p(CP_k)$$

(Lem. 6.1: output $\equiv$ input):

$$= in_0(CP_j) \sqcup \ldots \sqcup in_n(CP_j) \sqcup in_0(CP_k) \sqcup \ldots \sqcup in_p(CP_k)$$

(Eqn. 3.7: input history = filter state):

$$= fs_n(CP_j) \sqcup fs_p(CP_k)$$

■

6.3.2 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. This means that should any non-leaf channel or filter state be lost, the information necessary to regenerate that state exists at the leaves of any subtree that totally contains the lost components. To aid in our discussion, we introduce a new operator, $desc^k$, which describes the set of descendants of a communication process that is k levels away:

$$\begin{array}{lll}
desc^0(CP_i) & \rightarrow & CP_i; \\
desc^1(CP_i, j) & \rightarrow & j^{th} \text{ child of } CP_i; \\
desc^1(CP_i) & \rightarrow & \{desc^1(CP_i, 0), \ldots, desc^1(CP_i, fanout(CP_i) - 1)\} \\
desc(\{CP_m, \ldots, CP_n\}) & \rightarrow & desc^1(CP_m) \cup \ldots \cup desc^1(CP_n); \text{ and} \\
desc^k(CP_i) & \rightarrow & desc(desc^{k-1}(CP_i)), 1 < k \leq tree\_depth
\end{array}$$

The fs and cs operators without subscripts are shorthand for the specified process' or channel's current state based on filtered or incident packets. Similarly, without subscripts, *in* and *out* designate the specified process' input and output history, respectively. Lastly, when any of these operators are applied to a set of processes or channels, they return the join of that operator applied to the individual processes.

Lemma 6.3 (The All-encompassing Leaf States) The join of the states at the leaves of a TBŌN subtree equals the join of the state at the subtree's root process and all the TBŌN in-flight data.

Proof of All-encompassing Leaf States Lemma

From Lemma 6.2, we deduce:

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

Substituting the former identities into the latter:

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

■

6.3.3 TBŌN Output Dependence

Finally, we show that the TBŌN computation's output stream is solely a function of the root process' filter state and the TBŌN's channel states. The TBŌN input is the stream of inputs filtered by the TBŌN leaf processes, $in(desc^k(CP_0))$, where k is the TBŌN depth. We define the effective TBŌN output, $out(CP_0)$, to be 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 6.1 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 6.4 (TBŌN Input/Output Equivalence) The input to a TBŌN's leaves is equivalent to the effective output at the TBŌN's root process: $in(desc^k(CP_0)) \equiv out(CP_0)$.

We now demonstrate the last of our fundamental TBŌN properties:

Lemma 6.5 (TBŌN Output Dependence) The output of a TBŌN computation is solely a function of the TBŌN root process state and the TBŌN channel states.

Proof of the TBŌN Output Lemma

(By Cor. 6.4: Input/Output Equivalence)

$$out(CP_0) \equiv in(desc^k(CP_0))$$

(Eqn. 3.7: input history = filter state)

$$\equiv fs(desc^k(CP_0))$$

(By Lem. 6.3: All-encompassing Leaf State)

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

■

6.4 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, 6.3, 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 6.5, 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 the orphaned descendants of processes in a *failure zone*. 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

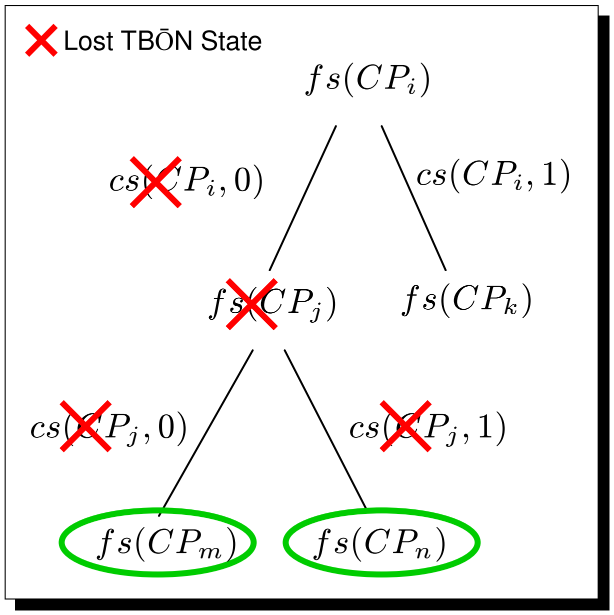


Figure 6.5 State Composition: When CP_j fails, $fs(CP_j)$, $cs(CP_j)$, and $cs(CP_i, 0)$ are lost. The circled states, $fs(CP_m)$ and $fs(CP_n)$, can be used to compensate for this lost state. (Only the TBŌN's filter and channel states are shown for simplicity.)

states used for compensation form a composite equivalent to the state that has been lost. It follows that state composition preserves a computation's semantics across failures:

Theorem 6.6 (State Composition) For idempotent aggregation operations, a TBŌN can tolerate failures without changing the computation's semantics by re-introducing filter state from the children of failed processes as channel state.

Proof of State Composition Theorem

Consider the TBŌN in Figure 6.5. 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 6.5, 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 6.3 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 **all** 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 it took to filter the original data that constituted the aggregate.

In Figure 6.6, we use our previous integer union computation to demonstrate the basic state composition mechanism and the concept of convergent output recovery. The left column of the figure shows the last four timesteps of the successful execution, and the right column of the figure shows an execution that uses state composition to recover from the failure of the process marked with an "x." In the right column, at timestep t_3, the TBŌN has been reconfigured to reconnect the orphaned processes to the root, and the former orphans have propagated their filter state to their new parents. In the following timesteps, t_4 through t_6, the reconfigured TBŌN resumes normal operation, continuing to propagate and filter input data until all data has been consumed by the TBŌN and propagated to the root process. Comparing the output of the two execution sequences, we observe a temporary divergence in their output streams at timesteps t_4 and t_5. However, at timestep t_6, the output stream of the failed TBŌN has converged to that of its non-failed equivalent. Additionally, we observe that in both cases, the sets of integers output at the root are equal.

6.4.1 Root Process Failure

State composition guarantees that all input data (in aggregate form) will be propagated eventually to the front-end process. During normal system operation, processes do not track explicitly what messages have been transmitted or filtered. This means that if the root process fails, we cannot know what output has already been received by the application front-end. Therefore, we must act conservatively and regenerate the entire TBŌN output stream: this entails a composition of filter states from all the children of the new root process.

As shown in Figure 6.7, when the root process fails, one of its children is promoted to the root position, and the remaining orphans become descendants of the new root. As descendants, the previously orphaned processes may be direct children of the root, as in the figure, or placed further down in the tree to keep it balanced. Normally, state composition is a distributed process: the orphans simply propagate their filter state to their new parent. Root process failure leads to a special case in which state composition is centralized at the new root. In this case, orphaned processes transmit their filter state

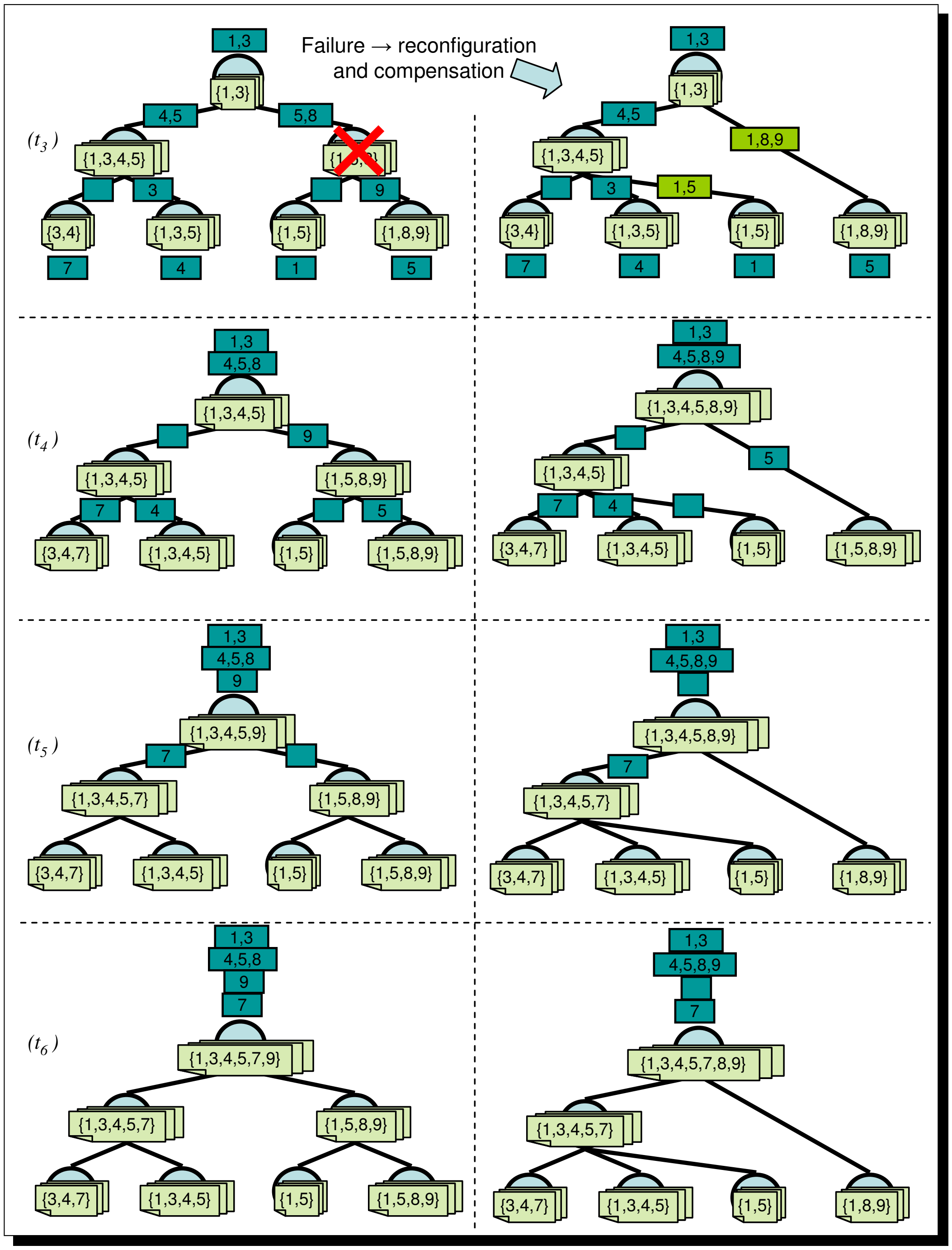


Figure 6.6 State Composition Example. The left column shows our previous integer union example. The right column shows this computation after a failure recovery at t_3. After a temporary output divergence at t_4 and t_5, the output of the failed TBŌN converges to that of its non-failed equivalent.

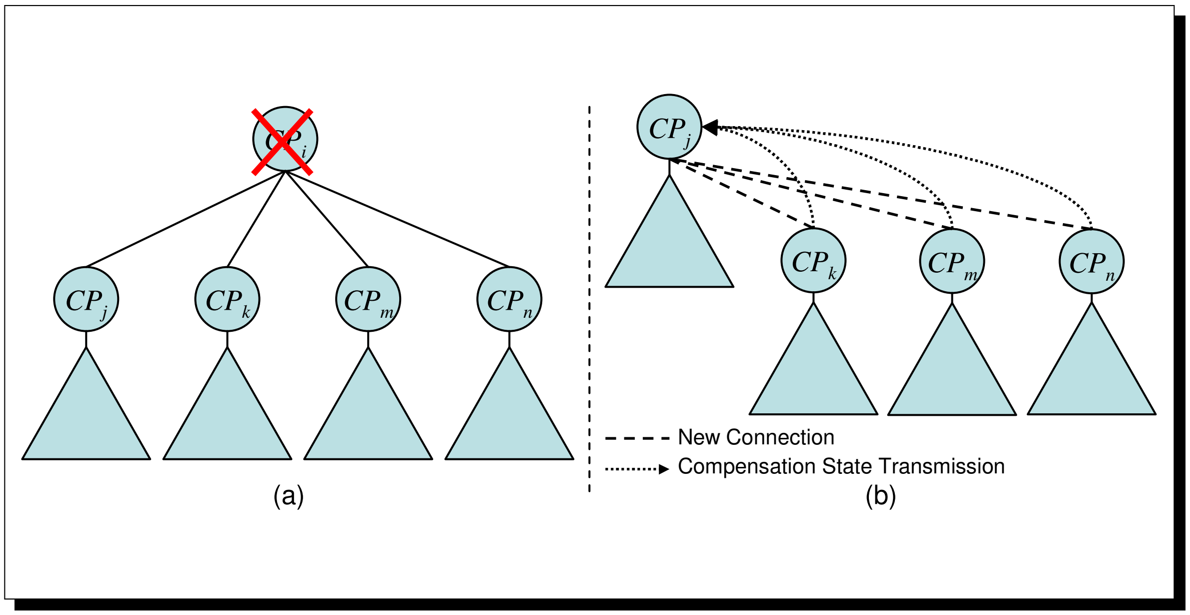


Figure 6.7 Composition for Root Process Failure: when the root process fails, one of its children becomes the new root. The new root orchestrates the state composition process.

to the new root process. These filter states encapsulate the entire input history for the subtrees rooted by the orphaned processes. The new root merges these filter states with its own resulting in a composition based on the entire input history of the original root process' 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.

6.4.2 Leaf Process Failures

In many situations, application back-ends, which connect to the TBŌN leaf processes, aggregate data from multiple sources. For example, in the case study presented in Chapter 5, each STAT back-end collects and aggregates stack traces from all collocated application processes. Therefore, in our model, filters are executed in the application back-end process for aggregation of 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 into the TBŌN. We discuss back-end process failures in Section 6.6.

6.4.3 Overlapping Failures

We define two failures as *overlapping* if the second failure occurs before recovery from the first completes. Our failure recovery model has two phases, tree reconfiguration, during which disconnected subtrees are re-connected to the application front-end and state compensation, during which our state recovery mechanisms compensate for any state that may have been lost due to the failure(s). For state composition, overlapping failures occur either when another failure occurs during the reconfiguration phase of the first or another failure occurs during the compensation phase of the first. There is a third scenario in which the two failures happen simultaneously, but in practice, we cannot distinguish this from the first scenario. During failure recovery, an orphaned process only interacts with its new parent. Therefore, we only need to consider overlapping failures relative to an orphan or its new parent, as shown in Figure 6.8.

First, we consider cases in which a second failure occurs during the reconfiguration phase of a previous failure. During reconfiguration, orphans attempt to connect to new parents that will reconnect the orphans to the root. If during this operation an orphan

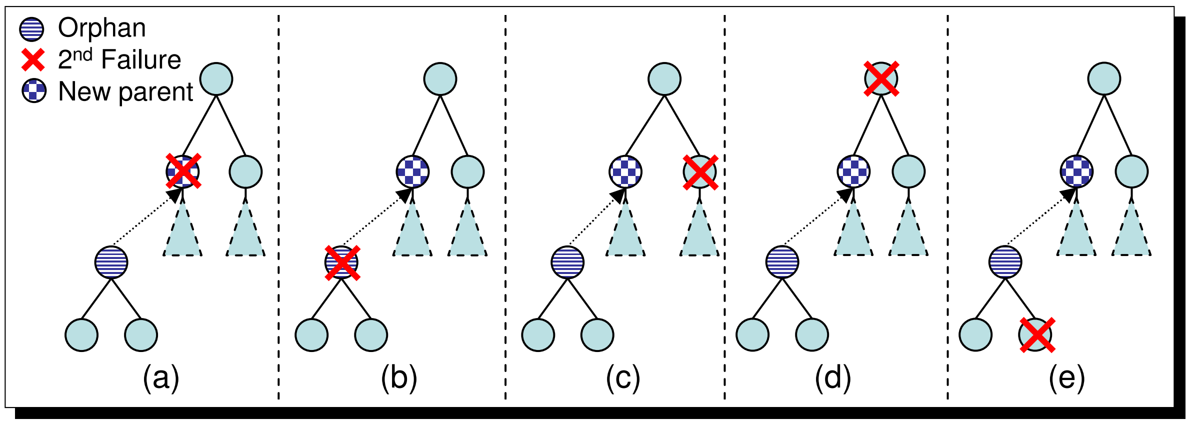


Figure 6.8 Overlapping Failures: (a) the adopting parent fails, (b) the orphan fails, (c) a process other than the orphan or adopting parent fails, (d) the parent of the adopting parent fails, and (e) a child of the orphan fails.

detects the failure of its adopting parent, Figure 6.8a, the orphan initiates another reconfiguration to choose a different parent for its adoption. If the orphaned process itself fails during reconfiguration, Figure 6.8b, its children become orphans and each initiates its own failure recovery process to reintegrate into the TBŌN; as in the normal case, the parent of the failed process does not actively participate in the failure recovery. If a process other than the orphan or the adopting parent fails, Figure 6.8c–e, the original reconfiguration is not affected, however, each of the newly disconnected orphans performs its own failure recovery operation. If the parent of the adopting process fails, Figure 6.8d, after the original reconfiguration completes, the adopting parent initiates a failure recovery for its parent's failure.

When two orphan's are performing a tree reconfiguration at the same time, we must give special consideration to the improper formation of cycles. For example, in Figure 6.9, at the time the orphans initiate their reconfiguration, the both determine that

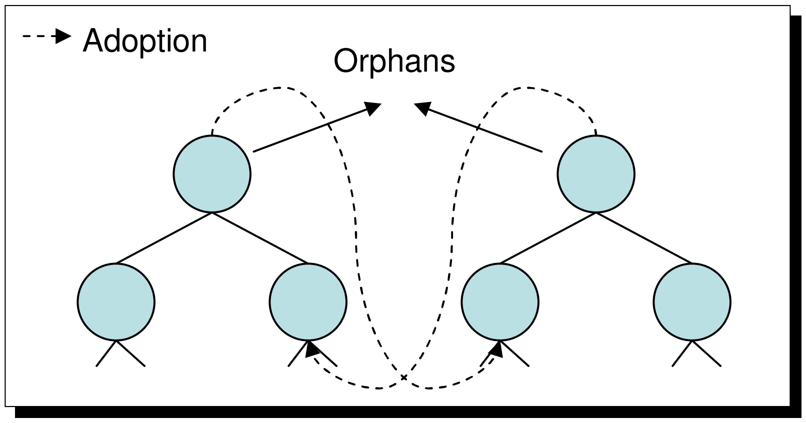


Figure 6.9 Overlapping Reconfiguration can cause Cycles: if the shown adoptions take place at the same time, a cycle is produced and we no longer have a TBŌN.

the shown adoption will not produce a cycle. However, if both adoptions occur, a cycle is produced. In Chapter 7, we describe tree reconfiguration algorithms in which orphaned processes make independent decisions in determining their new parent. After an orphan determines its new parent, we use a transaction commit protocol between the orphan, the new parent, and the new parent's ascendants to guarantee that the new parent and its ascendants will not perform a simultaneous reconfiguration that would result in a cycle.

For state recovery using state composition to compensate for potentially lost state, orphans transmit their filter state as output to their new parent. Since we are compensating for lost channel state, recovery is complete as soon as the filter state is sent by the orphan, even if the state has not been received by the new parent. (If the parent fails before receiving the compensating state, the child will be orphaned once again and send its compensating state to its new parent after yet another reconfiguration.) Practically, this means the state recovery phase is atomic: either (1) the filter state has not been sent and we can treat the system as though it were still in the reconfiguration phase as above, or (2) the filter state has been sent and recovery is complete such that a second failure does not overlap with the first.

6.5 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. State decomposition addresses non-idempotent computations by precisely calculating lost information and compensating for only that information, thereby removing any potential for re-processing the same input data multiple times. Intuitively, the parent of the failed process should filter the same input information as the surviving processes directly below it, namely, the children and siblings of the failed process. As an example, consider the TBŌN subtree in Figure 6.10 in which process CP_k has failed and the the channel states, $cs(CP_i, 1)$, $cs(CP_k, 0)$ and $cs(CP_k, 1)$ are lost. If CP_i updates its state with the inputs from its surviving channel, $cs(CP_i, 0)$, then the lost state information is the difference between the filter state (input history) of CP_i, the parent of the failed process, and the filter states of CP_j, CP_m, and CP_n, the siblings and children of the failed process. In other words, the set of processes, $\{CP_i,\ CP_j,\ CP_m,\ CP_n\}$, can be used to recover the state lost from CP_k's failure. The set of processes that participate in another process' failure recovery is called that process' *recovery clique*. Formally, for any

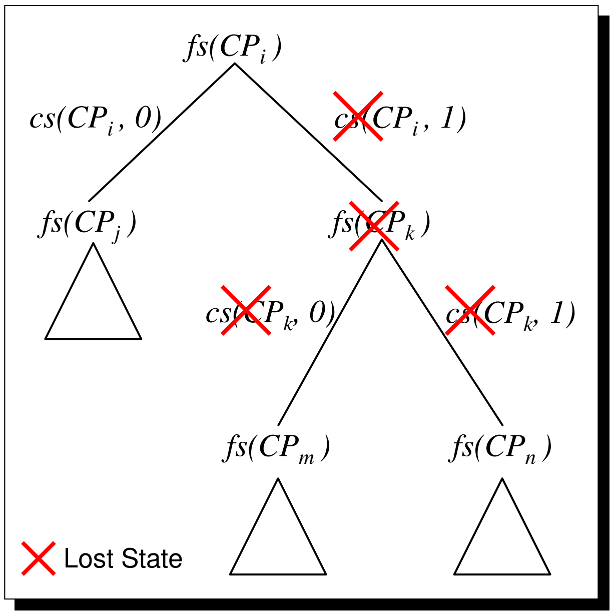


Figure 6.10 State Decomposition: When CP_k fails, $fs(CP_k)$, $cs(CP_i, 1)$, $cs(CP_k, 0)$, and $cs(CP_k, 1)$ are lost. The surviving states can be used to precisely compute the lost channel information. The set $\{CP_i, CP_j, CP_m, CP_n\}$ form the state decomposition *recovery clique*

process CP_i, its recovery clique is the set of processes, RC:

$$RC = CP_i\text{'s parent } \cup CP_i\text{'s siblings } \cup CP_i\text{'s children.}$$

Theorem 6.7 (State Decomposition Theorem) For non-idempotent aggregation operations, a TBŌN can tolerate failures without changing the computation's semantics by re-introducing the lost channel states. Further, this lost state is the difference between the join of the filter state at the parent of the failed process and its surviving channels' states and the join of the filter states of the failed process' children and siblings. Using the subtree in Figure 6.10 as an example:

$$cs(CP_i, 1) \sqcup cs(CP_k, 0) \sqcup cs(CP_k, 1) =$$
$$fs(CP_j) \sqcup fs(CP_m) \sqcup fs(CP_n) - (fs(CP_i) \sqcup cs(CP_i, 0))$$

and when CP_i drains the pending input on its surviving channels:

$$cs(CP_i, 1) \sqcup cs(CP_k, 0) \sqcup cs(CP_k, 1) = fs(CP_j) \sqcup fs(CP_m) \sqcup fs(CP_n) - fs(CP_i)$$

Proof of State Decomposition Theorem We prove the theorem for the binary subtree in Figure 6.10; the proof has a straightforward extension for subtrees with arbitrary fan-outs.

When CP_j fails, the states $fs(CP_k)$, $cs(CP_i, 1)$, $cs(CP_k, 0)$ and $cs(CP_k, 1)$ are lost. As in the proof of the State Composition Theorem, the All-encompassing Leaf State Lemma says that we only need to recover the lost channel states, $cs(CP_i, 1)$, $cs(CP_k, 0)$ and $cs(CP_k, 1)$. However, state decomposition must avoid over-compensation and must compute precisely the lost states. We now demonstrate how the lost channel states, $cs(CP_i, 1)$, $cs(CP_k, 0)$ and $cs(CP_k, 1)$, can be computed from the surviving states, $fs(CP_i)$, $cs(CP_i, 0)$, $fs(CP_j)$, $fs(CP_m)$ and $fs(CP_n)$.

(Lem. 6.2: filter state joined with channel state equals join of children's filter state.)
1. $fs(CP_i) \sqcup cs(CP_i, 0) \sqcup cs(CP_i, 1) = fs(CP_j) \sqcup fs(CP_k)$

(CP_i depletes it surviving channel, resulting in $cs(CP_i, 0) = \varnothing$.)
2. $fs(CP_i) \sqcup cs(CP_i, 1) = fs(CP_j) \sqcup fs(CP_k)$

(Lem. 6.2: filter state joined with channel state equals join of children's filter state.)
3. $fs(CP_k) \sqcup cs(CP_k, 0) \sqcup cs(CP_k, 1) = fs(CP_m) \sqcup fs(CP_n)$

('$-$' is inverse of '$\sqcup$')
4. $fs(CP_k) = fs(CP_m) \sqcup fs(CP_n) - (cs(CP_k, 0) \sqcup cs(CP_k, 1))$

(Substituting equality from line 4 for $fs(CP_k)$ into line 2.)
5. $fs(CP_i) \sqcup cs(CP_i, 1) =$
$$fs(CP_j) \sqcup (fs(CP_m) \sqcup fs(CP_n) - (cs(CP_k, 0) \sqcup cs(CP_k, 1)))$$

('$-$' is inverse operator of '$\sqcup$')
6. $fs(CP_i) \sqcup cs(CP_i, 1) \sqcup cs(CP_k, 0) \sqcup cs(CP_k, 1) =$
$$fs(CP_j) \sqcup fs(CP_m) \sqcup fs(CP_n)$$

('$-$' is inverse operator of '$\sqcup$')
7. $cs(CP_i, 1) \sqcup cs(CP_k, 0) \sqcup cs(CP_k, 1) =$
$$fs(CP_j) \sqcup fs(CP_m) \sqcup fs(CP_n) - fs(CP_i)$$

■

The result of the State Decomposition Theorem is that for TB$\bar{\text{O}}$Ns executing non-idempotent data aggregation operations, we can recover the information lost due to process failures. To compute this precisely, state decomposition relies on "$-$" being the inverse of "$\sqcup$", as opposed to the contextual inverse as in the case of state composition. The theorem motivates the basic failure recovery protocols demonstrated in Algorithms 6.1 and 6.2 to be performed by the parent and children of a failed process, respectively.

Like state composition, the failure recovery model for state decomposition entails a state recovery phase, executed by the parent of the failed node, and a tree reconfiguration phase, executed by the orphaned children of the failed node. The parent of the failed node

Algorithm 6.1: State decomposition algorithm executed at parent of failed process.

```
foreach Surviving Child do
    Pause child's input processing;
    Filter all pending input from child;
    Get child's filter state;
foreach Orphaned Process do
    Get orphan's filter state;
Use decomposition to compute lost channel information;
Propagate recovered information to parent;
foreach Surviving Child and Orphaned Process do
    Notify recovery complete;
Resume normal operation;
```

acts as the *orchestrator* of the state recovery phase during which the TBŌN compensates for its lost state. The orchestrator drains the input channels from its surviving children by instructing them to pause input processing and filtering all pending input from these channels. The orchestrator then retrieves the filter states from all its children and all the orphaned processes, performs the decomposition operation, and propagates the resulting compensating state to its parent. Finally, the orchestrator notifies the rest of the recovery clique that state recovery has been completed. During state decomposition, the orchestrator communicates with $O(fanout)$ processes.

Orphaned processes passively participate in the state recovery phase and actively participate in the tree reconfiguration phase. An orphan preempts the filtering of new

Algorithm 6.2: State decomposition algorithm executed by orphans.

```
Pause input processing;

Upon request, forward filter state to orchestrator;

Wait for “state recovery completion” notification from orchestrator;

Connect to a new parent to re-establish path to front-end;

Resume normal operation;
```

input data upon the detection of its parent’s failure. Upon request, the orphan forwards its compensating filter state to the orchestrator and awaits the orchestrator’s notification that state recovery is complete. At this point, the orphan executes a tree reconfiguration protocol, described in Chapter 7, to re-establish a path to the application front-end, and the failure recovery process completes.

6.5.1 Root Process Failures

State decomposition allows us to determine what input data the parent of a failed process will miss due to the failure based on that parent’s current state and the state of a few of its descendants. If the root process fails, state decomposition should determine precisely what input data has been received by the application front-end. If we execute the filter function at the front-end, the resulting filter state at the front-end will record the history of inputs that the front-end has received. Then if the root fails, a new root, chosen from amongst the previous root’s children, can orchestrate the decomposition process – using the filter state from the front-end to determine the lost information.

6.5.2 Leaf Process Failures

As we previously discussed, filters are executed also at the application back-ends to aggregate local data. As in state composition, the result is that the filter state at the back-ends can be used in the decomposition to compute the lost information when a TBŌN leaf process fails.

6.5.3 Non-overlapping Failures

State decomposition relies upon the TBŌN's inherent redundancy property, Lemma 6.2, which asserts the equivalence between a process' filter state and the filter states of its descendants. TBŌN reconfigurations change descendant relationships and, therefore, violate the inherent redundancy property for processes with changed ancestry. For example, a process that has adopted a new child would not have filtered the output that the child had sent to its former parent; in other words, the adopting parent and its new child would not have the same input history – the fundamental assumption of the inherent redundancy property. Consequently, a later decomposition that includes this parent and child in the same recovery clique will be erroneous. Therefore, after reconfigurations we must re-establish the inherent redundancy property to recover properly from *non-overlapping failures*, failures that do not overlap in time with previous ones.

As shown in Figure 6.11, when an orphan is adopted by a new parent, two branches of the root process change, the orphan's *old branch*, comprised of the orphan's former

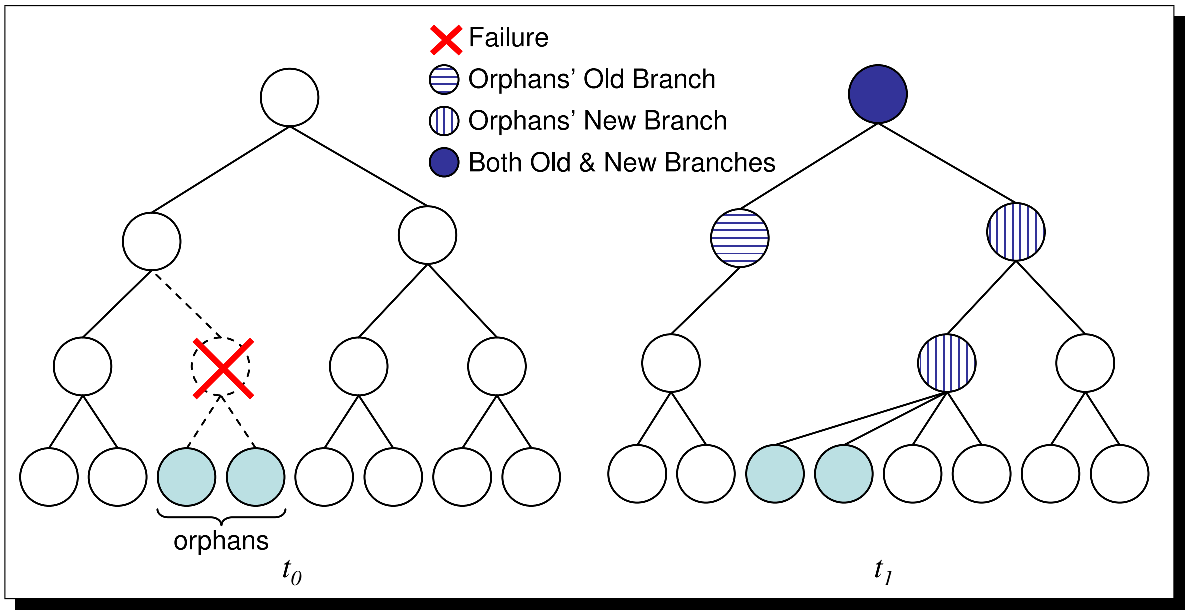


Figure 6.11 Reconfigurations and Branch Changes: t_0: the failure of the marked process results in two orphans. t_1: The orphans are adopted as shown, and two TBŌN branches change, the orphans' *old* branch containing the orphans' former ascendants, and the orphans' *new* branch containing the orphans' new ascendants.

ascendants, and the orphan's *new branch*, comprised of the orphan's new ascendants [2]. In this section, we discuss how TBŌN reconfigurations violate the inherent redundancy property and our mechanisms for re-establishing this property.

6.5.3.1 Non-overlapping Failures in the Orphan's Old Branch

We describe the impact that tree reconfigurations have on an orphan's old branch using Figure 6.12 in which CP_m has failed. At time t_0, the failed process' parent, CP_j, has orchestrated the state recovery and propagated the resulting compensating state to its parent. After the propagation event, CP_j signals the end of recovery, and the

[2]The TBŌN is reconfigured as one of the last steps of failure recovery, so ascendants common to both branches are not known, and we must consider all ascendants on the path to the root.

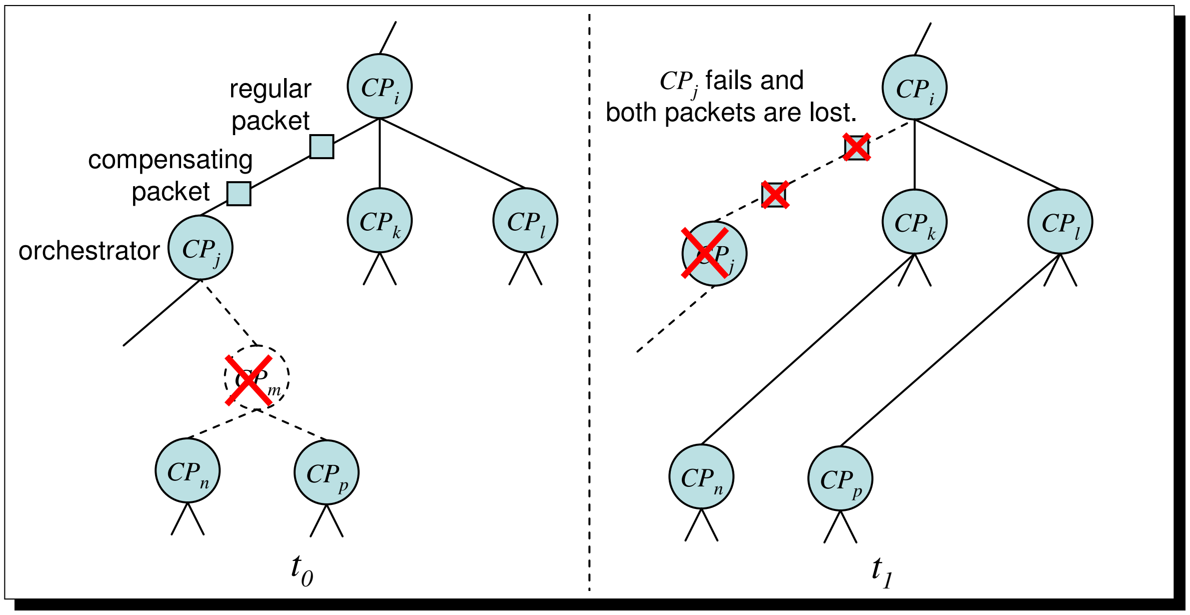


Figure 6.12 TB$\bar{O}$N Reconfigurations Violate Inherent Redundancy Property: at t_0, CP_j has compensated for CP_m's failure. At t_1, CP_j's failure before CP_i receives the compensating state, results in its loss; CP_n and CP_p are needed to recover this state. After t_1, CP_i's state still is dependent on that of its former subtree rooted at CP_j: $fs(CP_i) \neq fs(CP_k) \sqcup fs(CP_l)$.

topology is reconfigured such that CP_n and CP_p are adopted by CP_k and CP_l, respectively. If at time t_1, CP_j fails before its parent, CP_i, has received the compensating state, it becomes lost. Since CP_n and CP_p were no longer descendants of CP_j, if CP_i orchestrates a decomposition using the updated topology, the information lost from CP_n and CP_p would not be recovered. The change in CP_j's ancestry violated the inherent redundancy property for CP_j. Generally, if a process for which the inherent redundancy property has been violated fails and that failure results in information loss from that process' former descendants, the failure is not recoverable. We avoid this by having the orchestrator drain all channels on its path to the front-end before signaling the end of state recovery.

The channel drainage protocol removes all dependencies on the relocated subtrees' states from the channel states of their old branch from the root. However, the filter states of the relocated subtrees' former ascendants still are dependent on information from the relocated subtrees. For example, in Figure 6.12 after time t_1, CP_i has two remaining children, CP_k and CP_l, but when the channels between these processes are empty, the inherent redundancy lemma is violated: $fs(CP_i) \neq fs(CP_k) \sqcup fs(CP_l)$, since CP_i also contains state from its former descendants. This is the case for all former ascendants of the relocated subtrees. Therefore, we extend the state recovery protocol to re-establish the inherent redundancy property: the orchestrator and all ascendants remove the portion of their filter state contributed by their former descendants. This state is the join of the states that the orchestrator collected from its orphaned descendants for the decomposition. In Section 6.5.5, we present the final state decomposition algorithm including this and our other extensions and discuss the performance implications of the added extensions.

6.5.3.2 Non-overlapping Failures in the Orphan's New Branch

Just as removing a subtree from a branch of the root violates the inherent redundancy property in that branch, adding a subtree to a new branch also violates this property in the new branch. After a process adopts a new subtree, by default the adopter's filter state does not include any filter state from the adoptee. For example, in Figure 6.13, after CP_i adopts CP_j, CP_i's state does not include any of CP_j's state. If CP_j propagates this state to CP_i without special consideration, then the information in CP_j's state will be

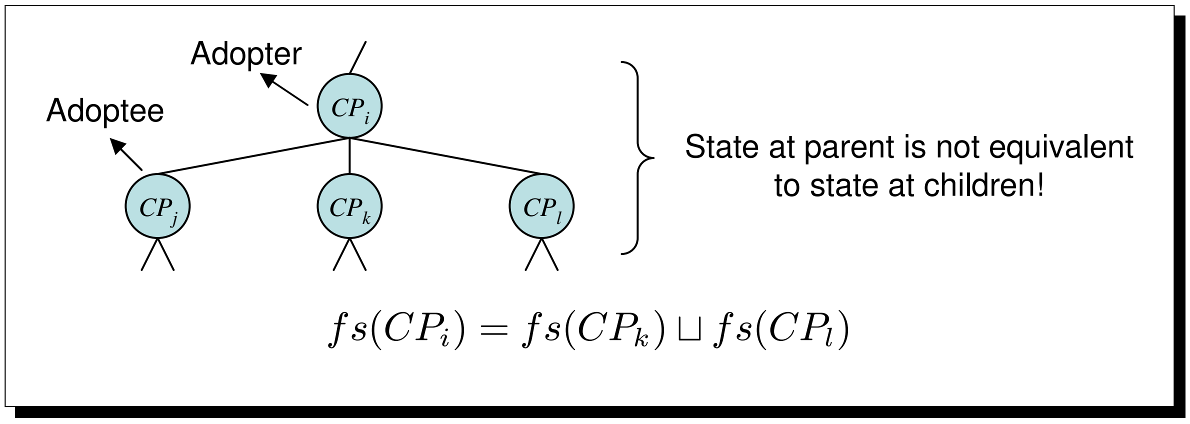

Adopter
Adoptee
CP_i
CP_j
CP_k
CP_l
State at parent is not equivalent to state at children!
$fs(CP_i) = fs(CP_k) \sqcup fs(CP_l)$

Figure 6.13 Adoption violates the Inherent Redundancy Property

filtered twice (once in the relocated subtree's former branch and once in the subtree's new branch), a violation for non-idempotent operations.

To re-establish the inherent redundancy property, the adopted process uses a special protocol to synchronize its state with its new ascendants. The adopted process sends its state to each of its new ascendants; upon receipt of this state, an ascendant merges this state into its current filter state but suppresses any resulting output, which would have already been propagated along the old branch. For ascendants common to an adopted process' old and new branches, the orphan's filter state is removed (as described in the previous section) and re-added. If the new reconfiguration were to be known earlier, this unnecessary state removal and re-addition could be eliminated.

6.5.4 Overlapping Failures

Failure recovery based on state decomposition entails interactions in both an orphaned process' old branch from the root and its new one. The failure recovery orchestrator in the orphan's old branch is responsible for recovering the lost information and re-establishing the inherent redundancy property in the old branch. The orphan is responsible for the tree reconfiguration and re-establishing the inherent redundancy property in its new branch. We now discuss how overlapping failures in these branches impact failure recovery.

6.5.4.1 Overlapping Failures in the Orphan's Old Branch

We distinguish two phases of the orchestrator's state recovery protocol, a state retrieval phase, in which filter state is gathered from the recovery clique, and an update phase, in which the compensating state (as well as the former descendants' states) is sent to all the orchestrator's ascendants. Therefore, in the orphan's old branch, an overlapping failure occurs when a previous failure recovery is in one of these two phases.

The parent of a failed process orchestrates state decomposition by first retrieving filter state from the default recovery clique, the siblings and children of the failed process. If the orchestrator fails during any phase of recovery, its parent will detect its failure and orchestrate a new failure recovery that encompasses the original failure zone.

If we have non-contiguous failures with disjoint recovery cliques, as in Figure 6.14a, the state retrieval process succeeds and failure recovery requires no special consideration. Our standard state decomposition mechanism recovers the lost information for each of the independent failure zones.

If the orchestrator fails to contact or receive the filter state from one of the recovery clique members, or the orchestrator itself fails, then we have a case of contiguous failures, Figure 6.14b. The closest surviving ascendant (CSA) of the failed processes expands the

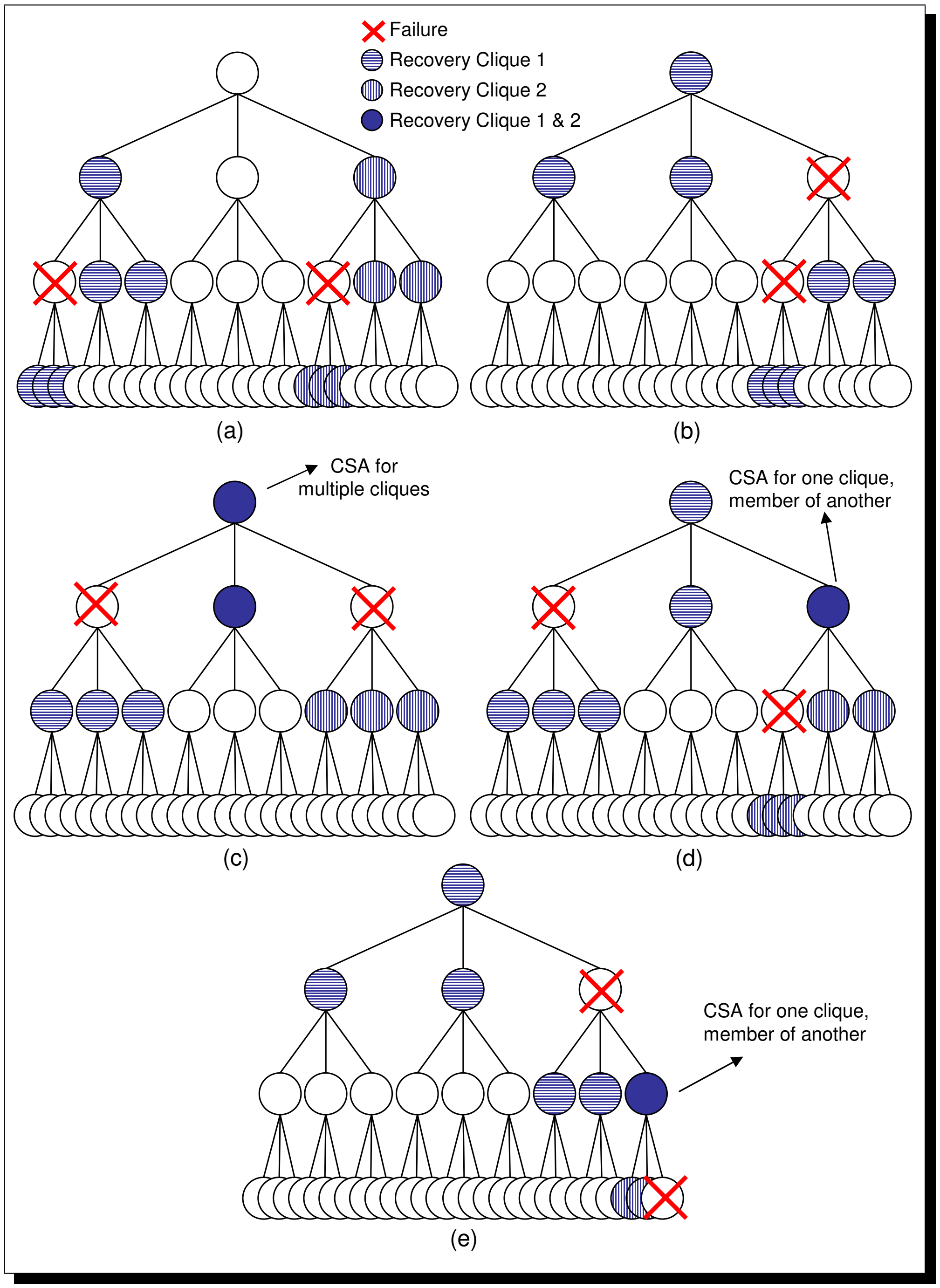


Figure 6.14 Multiple Failures and Recovery Cliques: (a) non-contiguous failures, non-intersecting recovery cliques, (b) contiguous failures, (c) non-contiguous failures, common closest surviving ascendant (CSA), (d,e) non-contiguous failures, intersecting cliques.

recovery clique to include the children of each contiguously failed process, descending each branch until it finds a complete set of non-failed children. Once state is retrieved from all the recovery clique members, state retrieval is complete, and failure recovery progresses as normal.

The remaining cases, Figure 6.14c-e, depict non-contiguous failures with intersecting recovery cliques. These cases only impact the latter update phase of the orchestrator's recovery protocol. If a process detects the failure of multiple children, Figure 6.14c, it orchestrates independent failure recoveries for each one. If the orchestrator of a failure is also a member of another recovery clique, Figures 6.14d and 6.14e, both failure recoveries may occur simultaneously with the intersecting member participating in both recovery cliques as prescribed.

The update phase completes when an orchestrator has drained the channels on its path to the front-end process and each ascendant on this path has removed the filter state from its former descendants. Therefore, the failure of one of the orchestrator's ascendants will impede the original failure recovery. We use a transaction commit protocol amongst the orchestrator and its ascendants to make the update phase atomic: either every process is successfully updated or none is. If one of the orchestrator's ascendant fails, the update phase fails, and the orchestrator must wait until its subtree is reconnected to the root before trying the update once again.

Normally, a parent detects and orchestrates the failure of its children. However, orphaned processes temporarily have no parent until the update phase of their parent's failure recovery process completes and the tree has been successfully reconfigured. To address this case, the orchestrator temporarily adopts its orphaned descendants and handles their failure.

6.5.4.2 Overlapping Failures in the Orphan's New Branch

For state decomposition, an orphan's failure recovery operation also has two phases, a reconfiguration phase, in which the orphan determines and connects to a new parent, and an update phase in which the inherent redundancy property is re-established in the orphan's new branch. TBŌN reconfiguration is independent of the compensation mechanism, so our previous discussion of overlapping failures during reconfiguration for state composition applies to state decomposition as well. We now discuss overlapping failures that occur during an orphan's update phase of failure recovery.

The orphan's update phase completes when all its ascendants have successfully updated their filter state with the orphan's filter state. Once again, we use a transaction commit protocol amongst the orphan and its ascendants to make this operation atomic. If the update operation succeeds, the orphan's failure recovery is complete. If one of the ascendants fail before the update completes, the orphan must wait until it is reconnected to the root and retry the update.

If the new parent of the orphan fails before the update completes, the update fails and the orphan initiates another reconfiguration to connect to a new parent that reconnects the orphan to the root. After this new reconfiguration, the orphan must perform an update to re-establish the inherent redundancy property in its new subtree. However, the failure of the update phase means that the inherent redundancy property is never reestablished in the new branch. As a result, the state of processes in the orphan's subtree and the other processes in the branch are inconsistent, and a decomposition that attempts to merge these inconsistent states will be erroneous. To avoid this circumstance, we extend the original update transaction to include the reconfiguration phase. Successful completion then means that the orphan has successfully connected to a new parent and updated its new branch with its state.

6.5.5 The Complete State Decomposition Recovery Protocol

The complete failure recovery protocol performed by the orchestrator is given in Algorithm 6.3. First, the orchestrator temporarily adopts all orphans and retrieves their filter states. Then the orchestrator instructs its surviving children to pause new input processing, drains the input channels to its surviving children and gathers their filter states. Using the filter states gathered from the entire recovery clique, the orchestrator then performs the state decomposition and propagates the result to its parent. The orchestrator then blocks until all the channels on its path to the front-end are drained. Finally, after atomically propagating the filter state from the orphans to all its ascendants, the orchestrator explicitly notifies its recovery clique members that the state recovery phase has completed.

The failure recovery algorithm executed by the orphaned processes is shown in Algorithm 6.4. When the orphan detects that its parent has failed, it pauses any further input data processing. Upon request, the orphan sends its filter state to the orchestrator

Algorithm 6.3: Orchestrator's State Decomposition Recovery Algorithm.

```
foreach Orphan do
    Temporarily adopt until recovery completes;
    Get filter state;
foreach Surviving Child do
    Pause child's input processing;
    Process all pending input from child;
    Get filter state;
Execute decomposition to compute lost channel information;
Propagate recovered information to parent;
Block until all channels on path to front-end are drained;
repeat
    BEGIN TRANSACTION
        foreach Ascendant on path to front-end do
            Send orphans' states to re-instate inherent redundancy property;
    END TRANSACTION
    if Transaction Fails then
        Block until reconnected to root;
until Transaction Succeeds ;
foreach Orphan and Surviving Child do
    Notify recovery complete;
```

Algorithm 6.4: Orphan's State Decomposition Recovery Algorithm.

```
Upon parent failure, pause further input processing;
Upon request, forward filter state to orchestrator;
Wait for notification of state recovery completion from orchestrator;
repeat
    BEGIN TRANSACTION
        Connect to a new parent to re-establish path to front-end;
        foreach Ascendant on path to front-end do
            Send filter states to re-instate inherent redundancy property;
    END TRANSACTION
    if Transaction Fails then
        Block until reconnected to root
until Transaction Succeeds ;
Resume normal operation;
```

and waits for notification of the completion of the state recovery phase. Once the state recovery phase is complete, the orphan atomically connects to a new parent and sends its filter state to its new ascendants. When the reconfiguration and update transaction completes, so is failure recovery, and the orphan resumes normal operation.

Several components of the state decomposition failure recovery protocols raise performance concerns. First, when a failure occurs, the orphans, the orchestrator and the orchestrator's surviving children preempt input processing. In our dataflow model, eventually each ascendant of the failed nodes stops processing new input data when there is no pending input from the orchestrator of the failure recovery. This potentially impacts the application's performance for the duration of failure recovery. In principle, temporarily orphaned processes can continue to filter new inputs, buffering outputs until the tree is reconfigured, and processes that have failed subtrees can continue to filter input data from their non-failed subtrees.

Second, the synchronous drainage of the channels on the orchestrator's path to the front-end will cause a delay depending on how much data was in-flight on those channels – potentially a lot in high throughput environments. However, to meet high throughput demands, the TBŌN should be able to filter and propagate data quickly during normal operation or failure recovery. Additionally, the drainage takes place automatically as the orchestrator's ascendants continue to filter their pending input concurrently with the orchestrator's failure recovery execution. As a result, the channels may be empty by the time the orchestrator needs them to be.

Third, the transactions executed by the orchestrator and the orphans have the potential for long delays. However, in each case, the transaction is between a limited number of participants, the processes on the path from the orchestrator or the orphan to the root. For the foreseeable future, this path is likely to be less than ten considering that a tree with a reasonable fan-out of 32 and depth of 6 would entail more than 10^9 processes.

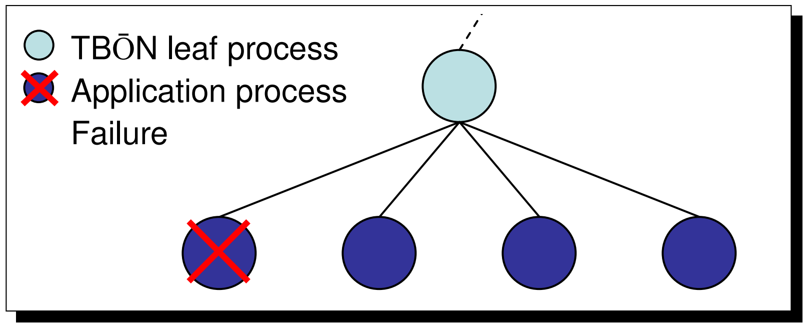


Figure 6.15 Application Back-End Failure

6.6 Discussion

In this chapter, we presented our state composition mechanism for reliable, idempotent data aggregation operations and our state decomposition mechanism for reliable, non-idempotent data aggregation. We now discuss some outstanding issues including the failure of the application end-point processes, the replacement of failed processes and various extensions to our TBŌN data aggregation model.

6.6.1 Application Back-End Process Failures

As shown in Figure 6.15, when an application back-end process fails, that process' state as well as the state of the channel that connected the back-end to the TBŌN is lost. Certain applications can tolerate the loss of some back-end input data and only require a process restart as opposed to a process recovery. For example, a data monitoring application simply may resume sending updated monitor data values, or even only report partial monitoring data for the surviving back-end systems if a restart is not possible.

Some applications cannot tolerate any data loss; for example, a debugging application performing an anomaly detection may lose valuable information pertaining to anomalies if some of its input data are missing. In these cases, if the back-end has no I/O channels other than to the TBŌN, the back-end processes may be viewed as sequential data sources amenable to light-weight, individual checkpoint protocols, which do not have the complexity and cost of distributed checkpoint protocols. In these cases, we can use checkpoint protocols that capture process and channel state [68, 105] such that a new incarnation of a failed process can resume the I/O channels of the original reliably.

6.6.2 Replacing Failed TBŌN Processes

Our failure recovery model does not require failed processes or hosts to be replaced before the TBŌN can resume normal operation. Instead, we reconfigure the TBŌN to re-route input data around failed components. However, our model does support dynamic topologies, which allow additional communication processes to join running TBŌN instantiations. These additional processes may be replacements for failed ones or new processes due to the availability of additional resources.

When a communication process detects the failure of its parent, the orphaned process dynamically connects to a new parent. New communication processes can join the TBŌN using this same dynamic connection capability. Then, a simple protocol can be used to tell select child processes to change their parent to the newly added process. We can avoid message loss without any compensation mechanisms by ensuring that a child process' original parent has processed all messages sent by that child before the child changes its parent. Alternatively, the child can change its parent at any time without flushing its output channel. In this latter case, a composition or decomposition is necessary upon reconnection to guarantee that no channel state information has been lost.

6.6.3 Directed Acyclic Graphs

While a formal analysis of our failure recovery model applied to directed acyclic graph (DAG) environments is left for future work, we hypothesize that state compensation can be used in such environments. The difference between TBŌN data aggregation and DAG data aggregation is that in the former each process sends its aggregation output to its single parent process. In the latter, process may send its aggregation output to multiple immediate successors. In a DAG environment, when a process' successor fails, after DAG reconstruction or process replacement, the processes that preceded the failed component can perform a state composition for idempotent aggregations and propagate their filter state to their new successor. Once again, the reconstruction process must be aware of the composition of aggregation operations to preserve the computation's semantics across reconfigurations. For non-idempotent aggregation operations, we must compute state decomposition recovery cliques for every immediate successor of the failed process to identify the information those processes would miss due to the failure.

6.6.4 Non-stateful Aggregations

Our failure model relies on the inherent redundancy of information found in the filter state for stateful aggregation operations. This redundancy does not exist in non-stateful aggregation operations. To make such operations reliable, we would need to introduce explicit data replication. For example, we could introduce a sliding window message buffering scheme for multi-hop networks [4, 25, 36, 88, 98]. Prior work in this area has addressed reliable transmission of raw data. Our aggregation-based model would require extensions to this previous work that properly track packets based on sequence numbers even when multiple packets have been aggregated into a single one.

6.6.5 Compositions of Heterogeneous Functions

Our current data aggregation model supports TBŌN computations that are compositions of a single aggregation operation that executes at all TBŌN processes. Our failure recovery model relies upon the commutativity and associativity of the aggregation operation that comprises this composition. To support the composition of heterogeneous operations, different aggregation operations at different TBŌN processes, our failure recovery model requires that this heterogeneous composition also meet our commutativity and associativity requirements. However, in general, compositions of commutative and associative operations do not retain these properties. Consider the example:

$f = x + 2$ and $g = x * 3$.

$f \circ g = x * 3 + 2$, but

$x * 3 + 2$ is not commutative: $x * 3 + 2 \neq x * 2 + 3$, and

$x * 3 + 2$ is not associative: $(x * 3) + 2 \neq x * (3 + 2)$.

In such cases, our current state compensation mechanisms do not work; we reserve this problem as a topic for future work.

6.6.6 An Alternative to State Decomposition

For non-idempotent data aggregation operations, we considered an alternative to state decomposition that combines a vector timestamp based strategy [60] with message logging. This approach promises[3] to avoid the process coordination mechanisms of state decomposition. However, this approach introduces explicit data replication and run time overhead during normal operation – scenarios we avoid in this dissertation.

In this alternative approach, back-ends apply sequence numbers to their output packets, and each aggregated packet has a vector of sequence numbers tracking the sequence numbers of the back-end packets that have contributed to the aggregate. Each communication process maintains a vector timestamp with an entry for each of its descendant back-ends tracking the highest sequence number that ascendant has seen from the back-end. Each communication process also tracks the outputs sent to its parent, buffering output packets until they are acknowledged by the application front-end.

When a failure occurs, the parent of the failed process notifies its orphaned descendants of the sequence number for the last packet it received from the failed process, and each orphan retransmits the lost packets from its output buffer to the parent of the failed process. Using the sequence numbers, communication processes discard packets that they have already filtered, for example, packets that have been retransmitted multiple

[3]We say "promises" since we have not explored this solution completely, and there may be scenarios that force the process synchronization mechanisms we hope to avoid.

times due to multiple failures, retaining the exactly once data processing requirement of non-idempotent operations.

Chapter 7

TBŌN Reconfiguration

As TBŌN processes fail, we must reconfigure the TBŌN to re-establish a path between orphaned processes and the application front-end. The reconfiguration algorithms must yield efficient topologies that continue to meet our low-latency, high bandwidth requirements. However, the algorithms must also be fast and scalable to keep failure recovery latencies low. We evaluate the dissemination and management of TBŌN process information needed by several reconfiguration algorithms and compare their performance.

Tree reconfiguration has two phases. In the first phase, an orphan selects a new parent, and in the second phase the orphan establishes a connection to its chosen new parent. In this chapter, we evaluate the time and space overhead of new parent selection and the impact of new parent choice on the resulting TBŌN configuration. In Section 8.2.3, we evaluate the time it takes for an orphan to establish a connection with its new parent – this latency is independent of the method an orphan uses to select its new parent.

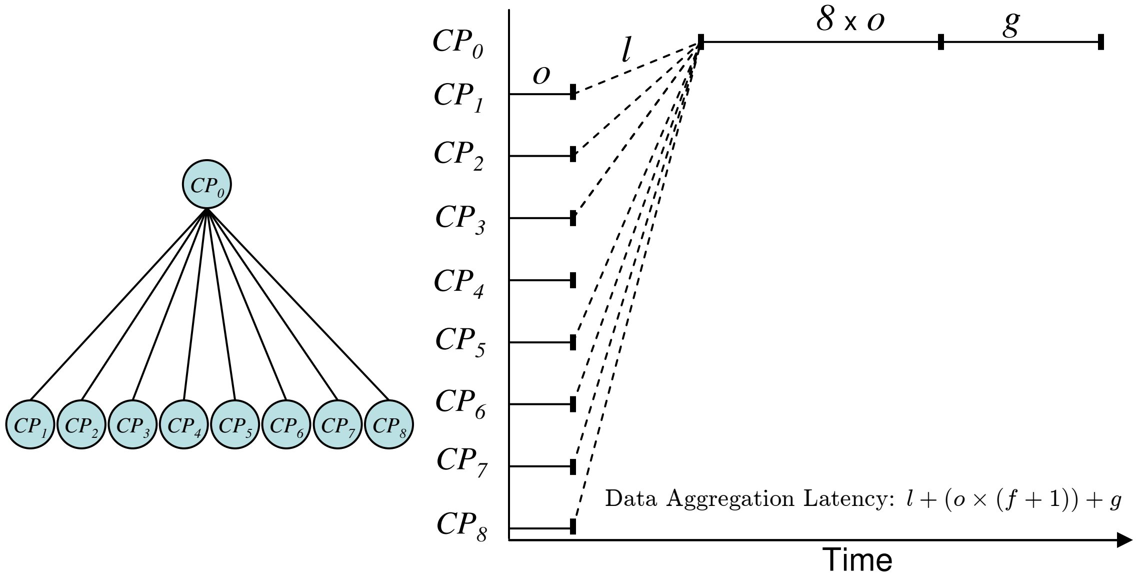


Figure 7.1 Data Aggregation Latency Model: o is message processing overhead, l is message transmission latency, f is TBŌN fan-out, and g is the inter-message gap, dominated by data aggregation execution time.

7.1 Characteristics of Efficient TBŌNs

The key characteristics that impact the latency and throughput of TBŌN data aggregation under normal operation are its fan-out and height. Figure 7.1 shows a time line for a single data aggregation operation at a communication process, CP_0, which has eight children, CP_1 through CP_8. In this figure, f is the fan-out. The other notation is adopted from the LogP model [27]: l is message transmission latency, o is message processing overhead, and g is the inter-message gap. In our model, g is dominated by the latency of the data aggregation operation.

Concurrently, the child processes transmit their output messages to CP_0. CP_0 must receive all f messages after which it can execute the aggregation operation. The latency of this entire operation can be modeled as $l + (o \times (f + 1)) + g$. Additionally, the fan-out typically impacts g, the latency of the aggregation operation.

For multi-level data aggregation, this process repeats h times, where h is the height of the tree. The overall data aggregation latency of a balanced TBŌN then can be modeled as: $h \times (l + (o \times (f + 1)) + g)$, demonstrating that TBŌN height is also a major factor in data aggregation performance. In fact, each component of the data aggregation latency function is multiplied by h. Additionally, based on our experiences, tree heights of less than five are reasonable, whereas fan-outs of 32-64 are also reasonable; so incremental increases in height have a greater effect on performance than incremental increases in fan-out. Therefore, minimizing height is more important than minimizing fan-out.

Before a parent process can execute its aggregation operation, all its children must complete their aggregation operations. The maximum fan-out of the child processes largely determines the delay before a parent can begin its data aggregation operation. A parent process only can be as fast as its slowest child; generally, the TBŌN only can be as fast as its slowest process. So the maximum fan-out of the TBŌN largely determines the TBŌN's overall performance. Lastly, keeping a TBŌN configuration balanced helps to avoid unnecessary bottlenecks at processes handling more than their fair share of the workload. We use standard deviation as the measure of the variability of TBŌN fan-outs because it captures the variability in the fan-outs as well as the mean deviation.

7.2 The Tree Reconfiguration Algorithms

We designed our algorithms along two dimensions: *adopter criterion* and *adopter sorting strategy*. The adopter criterion determines the processes that may adopt orphans and, therefore, the algorithm's data dissemination and management requirements. We require that orphans have the necessary process information, IP address, port and rank for each potential adopter, at the start of reconfiguration. Some algorithms additionally require a potential adopter's fan-out and subtree height.

Adopter sorting strategy determines how orphans rank their potential adopters and controls the complexity of the adopter selection process. We generate a sorted list so that should a potential parent fail by the time an orphan tries to connect to it, the orphan can attempt iteratively to connect to a new parent until an adoption succeeds.

The more process information that a reconfiguration algorithm uses, the greater the potential for yielding efficiently performing TBŌN topologies since a larger number of processes can be considered for adopting the orphans[1]. However, data dissemination and data management requirements also increase, and the time it takes for an orphan to select its new parent from amongst potential ones may increase as well.

7.2.1 Adopter Criteria

The adopter criterion determines the TBŌN processes that can adopt an orphan. We evaluate criteria that vary in data dissemination and data management requirements. For

[1]Technically, more choices also increase the potential for sub-optimal ones.

example, simple criteria like “root adopts” or “grandparent adopts” require each orphan to maintain information for a single process, but the performance of the resulting TBŌNs suffers. At the other end of the spectrum, the least restrictive criterion require that each process maintain information for every process in the tree. We also study criteria that maintain constant height while increasing fan-out, but not criteria that maintain constant fan-out while increasing height. This is because the negative performance impact of increased tree height is greater than that of increased fan-out,

During TBŌN instantiation for basic operation, each process receives topology information for its entire subtree. By default, each process manages information for $O(f^h)$ processes where f is the process’ fan-out and h is the height of its subtree. Based on the adopter criterion being used, we disseminate the extra process information needed for reconfiguration, and this information is updated as reconfigurations occur. We now describe our adopter criteria. After each description, for a TBŌN with fan-out f and height h, we specify (in parentheses) the total number of processes for which each TBŌN process must receive and store information.

- *root (RT)*: the root adopts all orphans. At instantiation, the root broadcasts its process information. (1 process).

- *grandparent (GP)*: the parent of the failed process adopts all orphans. At instantiation, each parent reports its parent to its children. (1 process).

- *root and children (RT+)*: the root or its children may adopt the orphans. At instantiation, the root broadcasts information for itself and its children. ($O(f)$ processes).
- *grandparent and children (GP+)*: the parent or siblings of the failed process may adopt the orphans. At instantiation, each parent sends to its children process information for its parent and other children. ($O(f)$ processes).
- *no height increase (NHI)*: any process may adopt an orphan as long as the adoption does not increase the tree's height; The root broadcasts process information for all processes. ($O(f^{h-1})$ processes).
- *unrestricted (ANY)*: any process may adopt an orphan; The root broadcasts process information for all processes. ($O(f^{h-1})$ processes).

We use the TBŌN's structure to efficiently disseminate the process information, and the majority of the communication overhead is incurred at TBŌN instantiation. Topology updates after reconfigurations are small in size: a node failure update requires only the failed node's rank information, and an adoption update requires only the adopter and adoptee's rank information. Furthermore, these updates are expected to be infrequent.

7.2.2 Sorting Potential Adopters

An orphan can use strategies of varying complexity to generate the keys used to sort potential adopters. More complex strategies that consider the TBŌN structure may yield better performing TBŌN structures, but the added complexity increases execution time and, therefore, failure recovery latencies. The adopter sorting strategies we consider are:

- *random (R)*: potential adopters are chosen randomly;

- *mapped (M)*: As shown in Figure 7.2, the potential adopters list and a list of the orphan and its siblings (also orphaned when the common parent fails) are sorted by rank. An orphan's index, r, in the orphan list maps it to an index, i, in the potential adopters list, where $i = r \% p$, and p is the number of potential adopters. Effectively, an orphan's sorted list of adopters are entries r through $p - 1$ then 0 through $r - 1$ in the sorted potential adopters list.

- *weighted random (WR)*: an *adoption weight*, a weight based on a process' fan-out and subtree height[2], is computed for each potential adopter. The weights are used to perform a weighted random sampling [29]: potential adopters are sorted using a key, $k_i = r_i^{\frac{1}{w_i}}$, where r_i is a random number uniformly distributed in $[0, 1]$, and w_i is the adoption weight for potential adopter, CP_i. For any two keys k_i and k_j, $P[k_i > k_j] = \frac{w_i}{w_i + w_j}$.

- *weighted mapped(WM)*: just like *mapped*, but the potential adopters are sorted by their adoption weight instead of rank.

We desire that orphans make tree reconfiguration decisions independently, without coordinating amongst each other, particularly since state composition does not use an orchestrating process. In a balanced tree when a process fails, orphans likely will compute similar adoption weights for each potential adopter, since the orphaned subtrees would

[2]The potential adopter's and the orphan's subtree heights determine height increases when using the unrestricted adopter criterion.

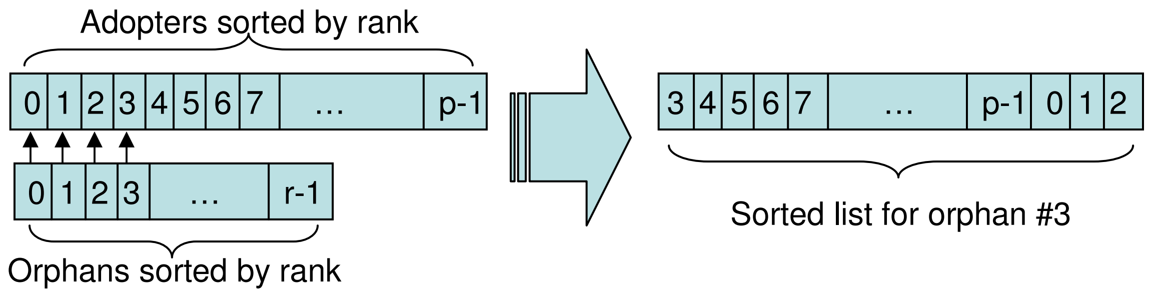


Figure 7.2 Mapped Sorting Strategy: an index in the sorted orphans list maps to an index in the sorted adopters list. An orphan's effective adopters list begins with the adopter mapped by its rank and wraps around the end of the original sorted adopters list.

have similar heights. This would lead to collisions in which all the orphans favor one or a small subset of the potential adopters. These sorting strategies use random or other distributions to mitigate orphan collisions.

Each tree reconfiguration algorithm is a composition of one adopter criterion and one adopter sorting strategy and is named accordingly. For example, the reconfiguration algorithm that considers any suitable parent and sorts the candidates randomly is named "unrestricted - random" or "ANY-R".

7.3 Evaluation

We evaluate the various tree reconfiguration algorithms' data management requirements and execution time as well as the characteristics of the TBŌN configurations produced by the algorithms. For our experiments, we use a version of MRNet extended to support our failure recovery model. For simplicity and to evaluate our algorithms at very large scales, we simulate the MRNet TBŌN; that is, we build and manipulate

MRNet topology data structures without actually instantiating the TBŌN process tree. We simulate failures by randomly deleting nodes from the tree and use the various algorithms to reconfigure the tree without the deleted node.

7.3.1 Data Requirements

The data requirements of a tree reconfiguration algorithm depend upon both the adopter criterion and the adopter sorting strategy. An orphan must collect and maintain information for every process that satisfies the adopter criteria: the more admissive the criteria the more information to manage. Adopter sorting algorithms that explicitly consider the TBŌN configuration must also manage additional meta-data that specify the process fan-outs and subtree heights.

The information required for each TBŌN process is a {rank, IP address, port} tuple, where the rank is a four byte identifier, the IP address is four bytes, and the port is two bytes for a total of ten bytes per process. In other words, a reconfiguration algorithm's data transmission and storage costs is ten bytes per process. Reconfiguration algorithms that consider the TBŌN configuration require an additional three bytes per process to represent fan-out (two bytes) and subtree height (one byte). Topology updates due to reconfigurations require four bytes to specify the rank of the failed process, and eight bytes to specify the ranks of adopter and adoptee processes for the orphan adoptions.

Table 7.1 shows the data management requirements for the various algorithms. We show the general formula, using f and h to specify fan-out and height, respectively, as well as the specific data requirements for a 32^4 TBŌN, one with a fan-out of 32 and a

Algorithm "adopter criteria - sorting strategy"			Data Requirement General	:	32^4 (1M) leaves
[GP, RT]	-	*	10	:	10 bytes
[GP+, RT+]	-	[R, M]	$10(f+1)$	:	330 bytes
[GP+ ,RT+]	-	[WR, WM]	$13(f+1)$	:	429 bytes
[ANY, NHI]	-	[R, M]	$10f^{h-1}$	:	320 KB
[ANY, NHI]	-	[WR, WM]	$13f^{h-1}$	:	416 KB

Table 7.1 Tree Reconfiguration Algorithm Data Management Requirements. A compressed view of the "adopter criteria v.s. sorting strategy" matrix. The column on the right gives the general data requirements as well as the specific requirements for a tree of fan-out, 32, and height, 4.

height of 4 totaling 1,048,576 leaf processes. The most demanding algorithms, the ones that consider (almost) all the parent processes and the TBŌN characteristics, require 416 kilobytes of data for our million process tree. The distribution of this data is a one time cost at TBŌN start-up, and 416 kilobytes is a small amount of data to maintain. We conclude that the dissemination and management of this amount of data is manageable.

7.3.2 Run Time Performance

A process that has become an orphan due to the failure of its parent performs a tree reconfiguration to determine the new parent to which it should connect. We wish to evaluate how each reconfiguration algorithm impacts the time an orphan takes to compute this new parent. To evaluate algorithm run time performance, we simulate

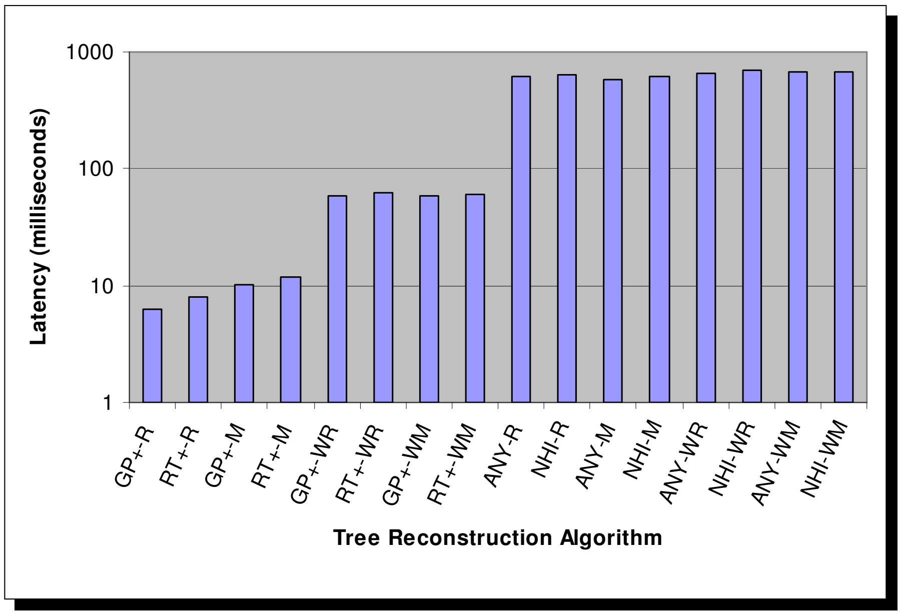


Figure 7.3 Tree Reconfiguration Latency: we evaluate the latency of our algorithms using a TBŌN with a fan-out of 32 and a height of 4, that is 32^4 or 1,048,576 leaves. The results are shown on a log scale.

random failures in our 32^4 TBŌN and measure the time it takes for an orphan to compute its new parent.

Naturally, there is no selection for the criteria that resolve to a single adopter, namely, *root* and *grandparent*. We evaluate the run time costs for the other criteria on an AMD Athlon 64 3800+ workstation with four gigabytes of memory. The results are shown in Figure 7.3. As expected, the adopter criteria has the biggest impact on algorithm run time: the less restrictive the criteria, the more potential adopters to consider, and the longer the algorithm takes. The NHI and ANY criteria consider most of the parents in the tree and have the longest latencies. The algorithms based on these adoption criteria execute in 600-700 milliseconds, still low enough to be suitable for our failure recovery model.

7.3.3 Tree Reconfiguration Algorithm Output

We also evaluate the tree reconfiguration algorithms' *output quality*, the quality of the resulting topologies. The output quality metrics we use are tree-height, maximum fan-out and standard deviation of fan-out; these metrics largely determine the new tree's data aggregation performance and load balance. After each reconfiguration, we record the output quality metrics of the resulting topology.

For each experiment, we simulate 128 failures for a TBŌN that starts with a balanced 32^3 tree[3]. Table 7.2 shows that for a significant fraction of the HPC systems presented in Chapter 1, a system of 32^3 processors is expected to experience 128 failures in less than 5-7 days – a reasonable run time for many HPC applications.

Height increases have a greater impact on TBŌN performance than fan-out increases. However, as a result of our initial balanced topologies, only the unrestricted adopter criterion can lead to tree height increases. Therefore, we first evaluate the impact of algorithm choice on maximum fan-out and standard deviation. Only if an algorithm based on the unrestricted adopter criterion significantly outperforms the others in these metrics, must we evaluate its height increase metric to observe the trade-off. On the other hand, any algorithm that can not result in height increases and performs as good as or better than those based on the unrestricted criterion becomes our preferred solution.

[3]We simulate a 32^3 tree instead of a 32^4 tree to reduce the simulation time and simplify the results presentation. Even so, for our 18 algorithms, we simulate more than 2,300 failures and 70,000 adoptions. At 32^4 cores, the majority of these systems are projected to have hundreds failures in a single day.

System	Processors	Processor MTBF (years)	MTB 128 Failures: 32^3 Processor System (days)
BG/L	131,072	2237.20	3189.76
Seaborg	6,080	243.03	346.51
Franklin	19,320	98.45	140.37
White	8,192	47.69	68.16
Purple	12,256	41.97	59.84
Jacquard	712	31.25	44.56
Bassi	888	28.10	40.06
PDSF	550	13.64	19.44
Cluster 1	6,152	7.58	10.81
Cluster 2	544	7.54	10.75
Cluster 3	1,024	7.38	10.52
Cluster 4	512	4.70	6.70
Cluster 5	2,048	4.38	6.24
Cluster 6	128	4.16	5.93
Cluster 7	4,096	3.59	5.12
Cluster 8	4,096	3.48	4.96
Cluster 9	512	3.41	4.86
Cluster 10	2,048	3.09	4.40
Cluster 11	328	3.06	4.36
Cluster 12	256	2.88	4.11

Table 7.2 Failure Rates for the Sample HPC System in Chapter 1: The last column shows projected time to 128 failures.

7.3.3.1 Maximum Fan-out

To evaluate the impact of algorithm choice on maximum fan-out, we first compared the algorithms that use the same adopter criterion. Then, we took the best algorithms for each adopter criterion and compared them against each other. These "best of" results are shown in Figure 7.4. The weighted mapped strategy that considers all possible adopters, *ANY-WM*, yields the best results for maximum fan-out. After 128 failures, the *ANY-WM* algorithm has increased maximum fan-out by 16% compared to 25% for the

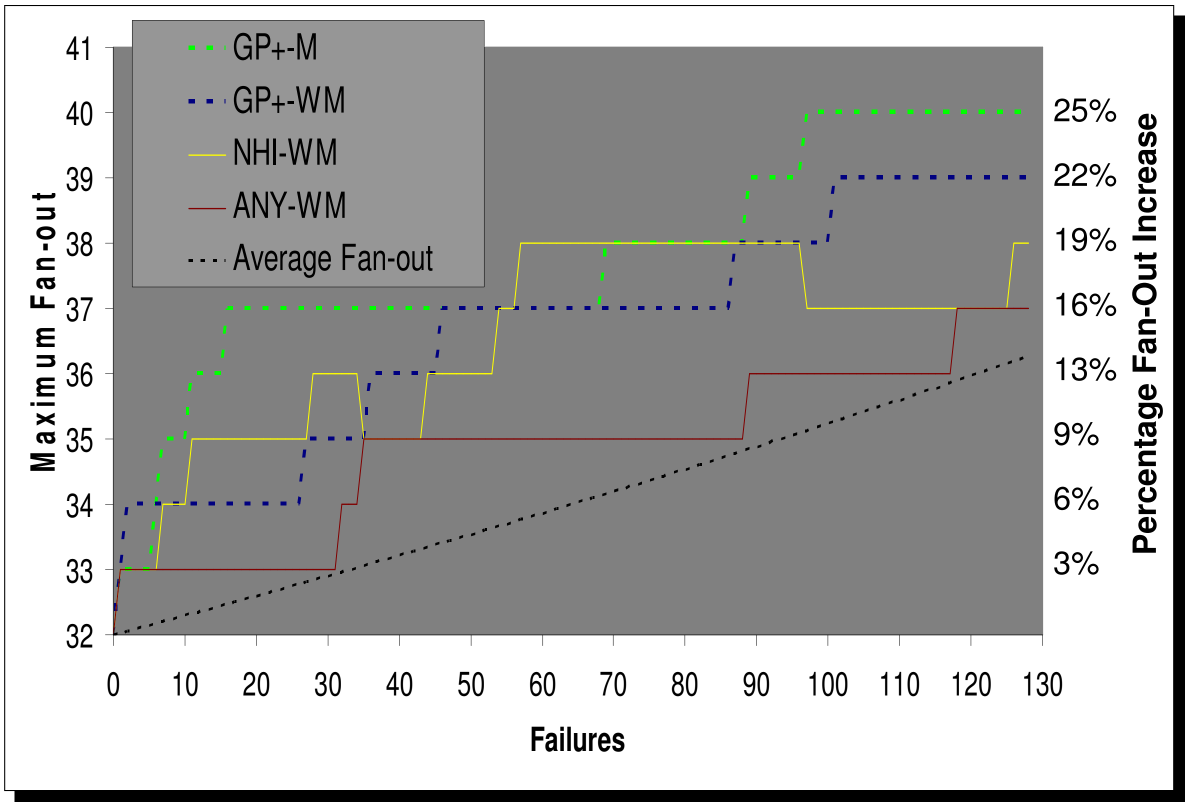


Figure 7.4 Best Algorithms for Maximum Fan-out: After 128 simulated process failures and tree reconfigurations in our 32^3 TBŌN, the *weighted mapped* sort strategy generally yielded the best results for maximum fan-out. The percentage increase in maximum fan-out, an estimate of the increase in TBŌN aggregation latency, is shown on the right.

worst performing algorithm, *GP+-M*. While *NHI-WM* is (sometimes) a close second, it can yield less favorable results. For example, if a node with a deep subtree fails, there may be only a few potential adopters since their subtrees must be as deep as that of the failed node to meet the no height increase restriction. If the failed node had many children, a few adopters adopt many orphans, and the fan-out can increase dramatically.

The *GP+-WM* and *GP+-M* algorithms perform surprisingly well considering that they require the dissemination and management of significantly less process information than *NHI-WM* and *ANY-WM*. Using the *GP+* adopter criterion, for each failure, orphans choose from a small set of potential adopters, and this set depends upon the location of the failure. With failures distributed uniformly throughout the tree, the potential adopters also become distributed uniformly throughout the tree. If the failures were not uniformly distributed throughout the tree, for example, if there were a concentrated area of failures, then the *GP+* adopter criteria would concentrate all the adoptions of orphaned nodes in this area resulting in greater increases in maximum fan-out.

7.3.3.2 Standard Deviation of Fan-out

Figure 7.5 shows the standard deviation of fan-out for the best tree reconfiguration algorithms. While for the *GP+-WM* and *GP-M* algorithms, the imbalance steadily grows, the *ANY-WM* and *NHI-WM* systematically correct the system's balance. This is because they deterministically choose the best potential adopters for orphans, so as nodes adopt orphans, they become less favored for the immediate future. Effectively, across multiple failures, orphans are distributed in a round-robin fashion to all the parents in the tree.

7.3.3.3 Height Increase

Given that an algorithm based on the *ANY* criterion indeed outperforms the others in terms of maximum fan-out and standard deviation of fan-out, we must evaluate how the use of this algorithm, *ANY-WM*, impacts tree height. *ANY-WM* deterministically chooses the better adopters that do not increase the tree's height, but as the fan-outs nearer to the root increase, eventually *ANY-WM* favors the nodes nearer the leaves of the tree with smaller fan-outs, and the height increases. As shown in Figure 7.6, throughout or

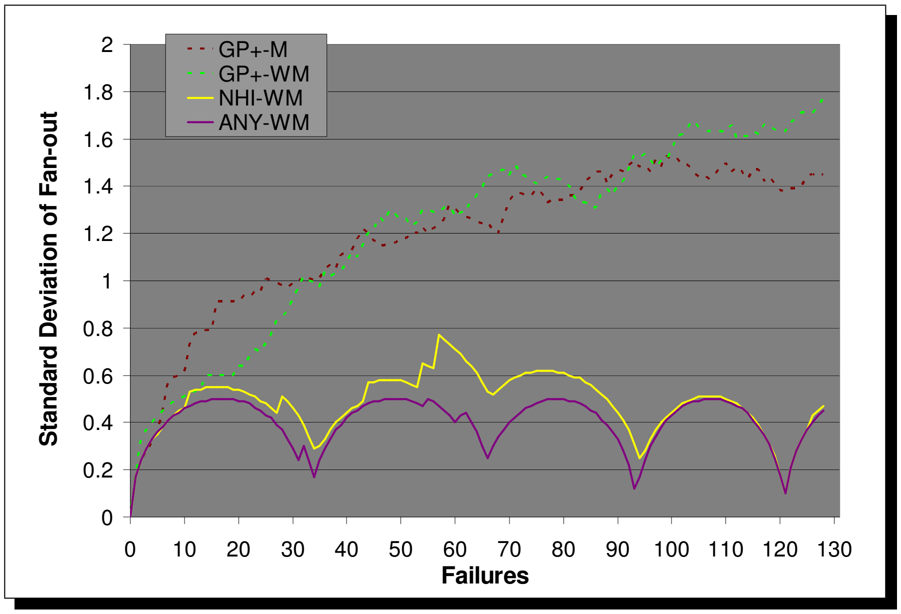


Figure 7.5 Standard Deviation of Fan-out for Best Algorithms: the *ANY-WM* and *NHI-WM* algorithms that consider the majority of the TBŌN processes as potential adopters keep the TBŌN fan-out well-balanced.

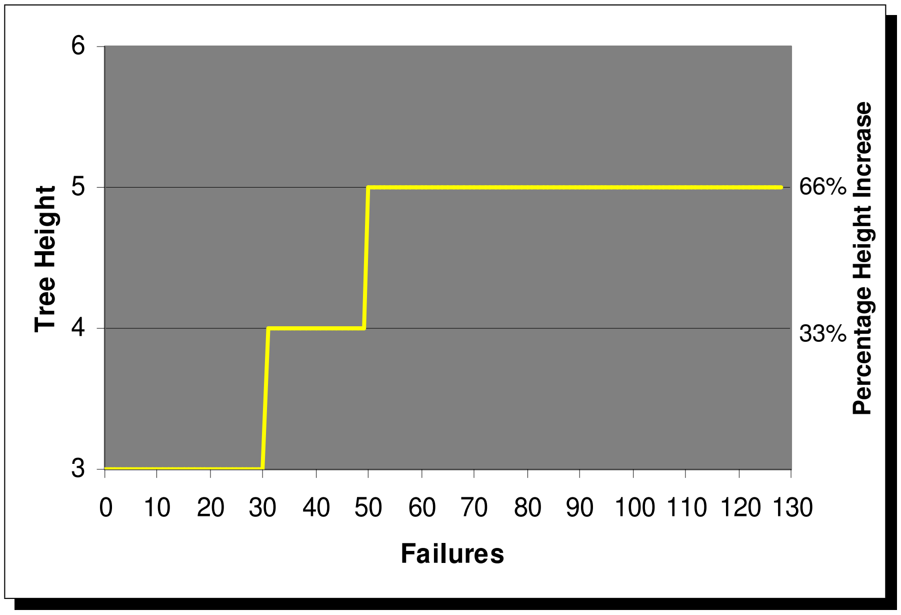


Figure 7.6 Height Increases for *ANY-WM* Algorithm. After the 128 simulated process failures and tree reconfigurations, the *ANY-WM* algorithm increased the TBŌN height, and, therefore, its aggregation latency, by 66%.

128 reconfigurations, tree height increased twice at failures 31 and 50. These increases represent 33% and 66% increases in tree height and, therefore, 33% and 66% increases in data aggregation latencies according to our performance model. In contrast, after 128 failures, the *NHI-WM* algorithm only increased maximum fan-out by 19%, so it should be favored over the *ANY-WM* algorithm.

7.4 Summary

Given that the execution time and data management costs of all the tree reconfiguration algorithms are more than acceptable at our target scales, we only need to consider the algorithms' output quality when choosing a tree reconfiguration algorithm. The best overall algorithm is *NHI-WM*: this algorithm moderately increases the tree's maximum fan-out and never increases its height. However, if information dissemination and management are still of concern and failures are expected to be uniformly distributed throughout the TBŌN, the *GP-M* and *GP-WM* algorithms have minimal data requirements and yield reasonable TBŌN configurations.

Chapter 8

An Experimental Study of State Composition

We extended MRNet with an implementation of our state composition failure recovery model. In this chapter, we describe the changes that we made to support failure recovery. We used this framework to validate empirically the state composition model as well as measure its performance and application perturbation. Our evaluation shows that the failure recovery latencies are very low resulting in unnoticeable application perturbation. More specifically, a TBŌN with a fan-out of 128 recovers from a failure in less than 80 milliseconds – with just 3 levels, a fan-out of 128 supports over two million application back-ends. Now, MRNet-based applications can leverage state composition for reliable operation at large scale.

8.1 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.

8.1.1 The MRNet Event Detection Service

Each MRNet process must detect important asynchronous system events like process failures or adoption requests. We use passive detection mechanisms that avoid active probing and its associated overhead. Our approach is to use connection-based mechanisms (TCP-based mechanisms, in the MRNet case) to notify a process that an interesting event has occurred.

The newly added 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 to establish *process peer* relationships. Process peers are two processes connected by a socket for direct inter-process communication. 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 an orphan to request an adoption.

8.1.2 Failure Detection

Failure detection in a distributed system comprises component failure detection and failure information dissemination. 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 for failures. Currently, an MRNet process monitors its parent and children. Therefore, the number of peers each process monitors is determined by the tree's fan-out.

Process peers monitor each other using *failure-detection event sockets*. When a process is adopted by a new parent, it sends the *New Failure Detection Connection* protocol message to its parent's EDS. Both the parent and child add their respective failure detection event sockets (the sockets used to send and receive the protocol message) to their watch lists. 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 a failure. Process crashes or terminations are detected immediately by peers: when a process fails, its host's kernel will abort its open connections, and these connection abortions will be detected immediately by the process' remote peers. Node or link failures prevent a kernel from explicitly aborting its connections. TCP keep-alive probes can be used to detect such failures; however, in general keep-alive probes have a two hour default period that can be lowered only on a system-wide basis by privileged users [105]. Heartbeat protocols [1, 2, 15, 22, 89, 44] could be used for more responsive, user controlled node and link failure detection.

When a process failure is detected, this failure information must be disseminated to all TBŌN processes so that they may update their topologies accordingly. Once again, we leverage the TBŌN structure for efficient, scalable failure information dissemination. Since each process is monitored by multiple peer EDSes, multiple 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 it to all its peers other than the one from which it received the report.

Reconfigurations and even normal 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 it has been adopted multiple times to different 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. Untimely (or late) failure reports lead to stale topology information. Our reconfiguration algorithms tolerate stale topology information by iteratively connecting to potential adopters, which may have failed, until an adoption succeeds. A missing failure report is infinitely late, and the above timeliness discussion holds. Different failure reports cannot contain conflicting information, so out-of-order failure reports do not pose any issue beyond that of stale topology information.

8.1.3 Dynamic Topology Configuration

Dynamic topology configurations are reorganizations of the TBŌN process tree that take place after the initial process tree had been established. Such reorganizations are necessary, for example, to accommodate process failures. After failures, orphans initiate reconfigurations using the algorithms described in Chapter 7. An orphan contacts a potential adopter's EDS with the *New Data Connection* protocol, and the adopter and adoptee establish a socket for application data transmission.

Like failures, reconfigurations modify the TBŌN topology and must be reported to all TBŌN processes. A reconfiguration report contains the adopter's and adoptee's ranks. Since disconnected sub-trees remain intact, this information is sufficient for recipients to update their topology information correctly. 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 it to all its peers other than the one from which it received the report.

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* [13] to address this problem. Each TBŌN 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.

As discussed in Section 6.4.3, the untimely delivery of a reconfiguration report may lead to erroneous cycles in the TBŌN topology 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 completely. Orphans perform a simple topology validation that can avoid some instances of cycle formation. In addition to cycle formations, late reconfiguration updates 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.

8.1.4 The New MRNet Instantiation

The newly added support for dynamic topology configurations allows us to replace MRNet's two previous instantiation modes described in Chapter 4, which supported only static topology configurations, with a single mechanism. The new mechanism supports both the case in which MRNet creates the application back-end processes as well as the case in which the application back-end processes are created by a third party mechanism.

In the new instantiation procedure, communication processes are instantiated as before – each parent process creates its child communication processes. However, children establish a data connection with their parent by sending the *New Data Connection* protocol message to the parent's EDS. Previously, a parent process would create a listening socket during the initialization phase to establish its child connections. Since the EDS implements a persistent service for dynamic configurations, now back-end processes can join the TBŌN session at any time, whether they were created by the MRNet infrastructure or another service.

Algorithm 8.1: 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;
```

8.1.5 State Composition Implementation

We derive a straightforward prototype of state composition from the theory presented in Section 6.4. As shown in Algorithm 8.1, 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 data aggregation is commutative and associative in our computational model, upon adoption it would be sufficient for an orphan to buffer and propagate the aggregate of its pending output instead of individual output packets.

After input data processing is paused, the orphan initiates the TBŌN reconfiguration. Using the *NHI-WM* algorithm detailed in Chapter 7, the orphan computes a sorted list of potential adopters and establishes a connection with the first surviving process from the list. After the reconfiguration, the adoptee and the adopter update their topology data to reflect the changed topology. 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 and the adopted processes and their adopters disseminate failure and reconfiguration reports as described in Sections 8.1.2 and 8.1.3.

8.1.5.1 Interface Details

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 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 to compensate for any lost state. We augmented the *load_FilterFunction* routine, which is used to load new filters into the MRNet infrastructure (See Section 4.4). When the new *load_FilterFunction* routine is invoked, it also queries 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 recoverable.

8.2 Evaluation

In the absence of failures, our failure recovery mechanisms do not incur any computational, network or storage resources beyond those necessary for normal TBŌN operation. In this section, we evaluate the time our implementation takes to recover from failures and the impact of failures on an application's 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 (DDR) InfiniBand network.

8.2.1 The Experimental Framework

The main component of our experimental framework is a failure injection and management service (FIMS) that injects TBŌN process failures and collects failure recovery performance data. The FIMS injects a failure by connecting to the EDS of a randomly chosen victim process and sending a special *TerminateSelf* message. The FIMS records the time each failure was injected.

After failure recovery, each previously orphaned process notifies the FIMS that its failure recovery is complete. The recovery completion message includes a performance breakdown of the individual failure recovery steps described in Section 8.2.3. The time from failure injection to the receipt of the last recovery completion message estimates the overall TBŌN failure recovery latency. The estimate is conservative because it includes the transmission delays of the *TerminateSelf* and recovery completion messages. Furthermore, notifications are received sequentially allowing for additional serialization delays.

8.2.2 The Application

We use the previously introduced integer union computation, 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 through the TBŌN at a rate of ten packets per second. After each experiment, we compare the input data generated by the back-ends with the output data produced at the front-end.

We use the integer union computation because its output is easily verifiable, and this computation is representative of more complex aggregations. For example, the sub-graph folding algorithm [77] used in the Paradyn tool performs essentially the same computation operating on graph data instead of integral data; that is, it performs union and difference operations on node and edge data instead of integers.

We empirically confirmed the functional correctness of the failure recovery model by using the FIMS to inject failures into running instances of this computation and validating the results. Had data been lost due to the injected failures, the output set at the front-end would be a proper subset of the union of the input sets.

8.2.3 Recovery Latency Micro-benchmark Experiments

Our failure recovery mechanisms may cause temporary divergences in the TB̄ON output: the associations and commutations of input data that have been re-routed due to failure may differ from that of the non-failed execution, or there may be some duplicate input processing. Eventually, the output stream converges back to that of the non-failed computation. The re-convergence occurs after all input data affected by failure have been propagated to the root process. The duration of the divergence can be estimated by:

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

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 the recovery latency and difference in propagation latencies between an orphan's old path to the root and its new path after reconfiguration. We focus on the orphans' recovery latencies, since technically, TB̄ON failure recovery is completed once each orphan has initiated the transfer of its compensating state to its parent. Recall that 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.

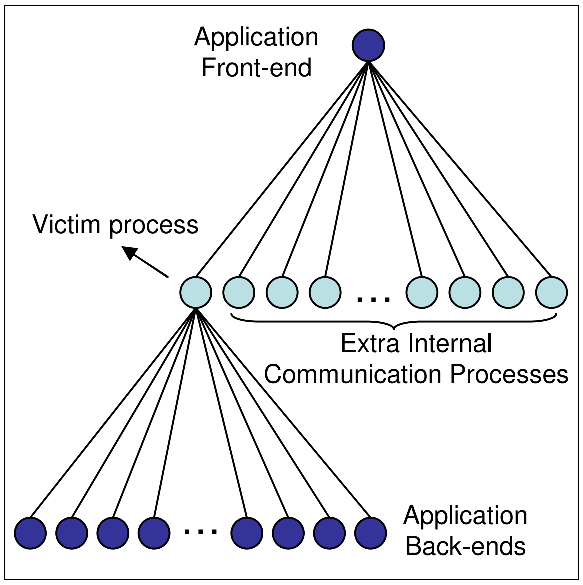
Each orphan's individual failure recovery latency is the sum:

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

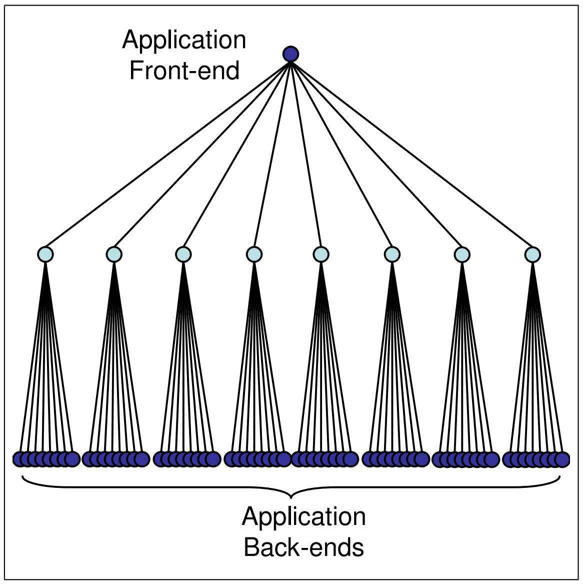
where $l(new\_parent)$ is the time to compute the new parent, $l(connect)$ is the time to connect to the new parent, $l(compensate)$ is the time to send the filter state[1], and $l(cleanup)$ is the time to update local data structures and propagate failure and reconfiguration reports. Recall that the FIMS also records a conservative estimate of the overall TBŌN failure recovery latency.

For state composition, orphaned processes are the primary actors of failure recovery, and the latency of failure recovery is a function of the tree's fan-out, not total size. Therefore, we evaluated the impact that the number of orphans caused by a failure has on failure recovery latencies. Our MRNet experiences suggest 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. Resource constraints did not allow us to test balanced trees with such large fan-outs, so we organized the micro-benchmark topologies with

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



(a) Micro-benchmark Topology



(b) Macro-benchmark Topology

Figure 8.1 Topology Organization for Experimental Evaluations. (a) Micro-benchmark Topology: The victim process has the fan-out being evaluated, and 16 internal processes are added to distribute the orphan adoptions. (b) Macro-benchmark Topology: the initial tree is balanced with two levels. Back-end processes are located eight per node (one per CPU), and parent processes are located one per node.

one TBŌN process per node such that only designated victim processes had the large fan-outs, as shown in Figure 8.1(a). We added 16 additional processes to distribute the orphan adoptions; this reflects practical TBŌN topologies in which orphaned processes have multiple potential adopters to choose from.

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 8.2. $l(new\_parent)$ 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

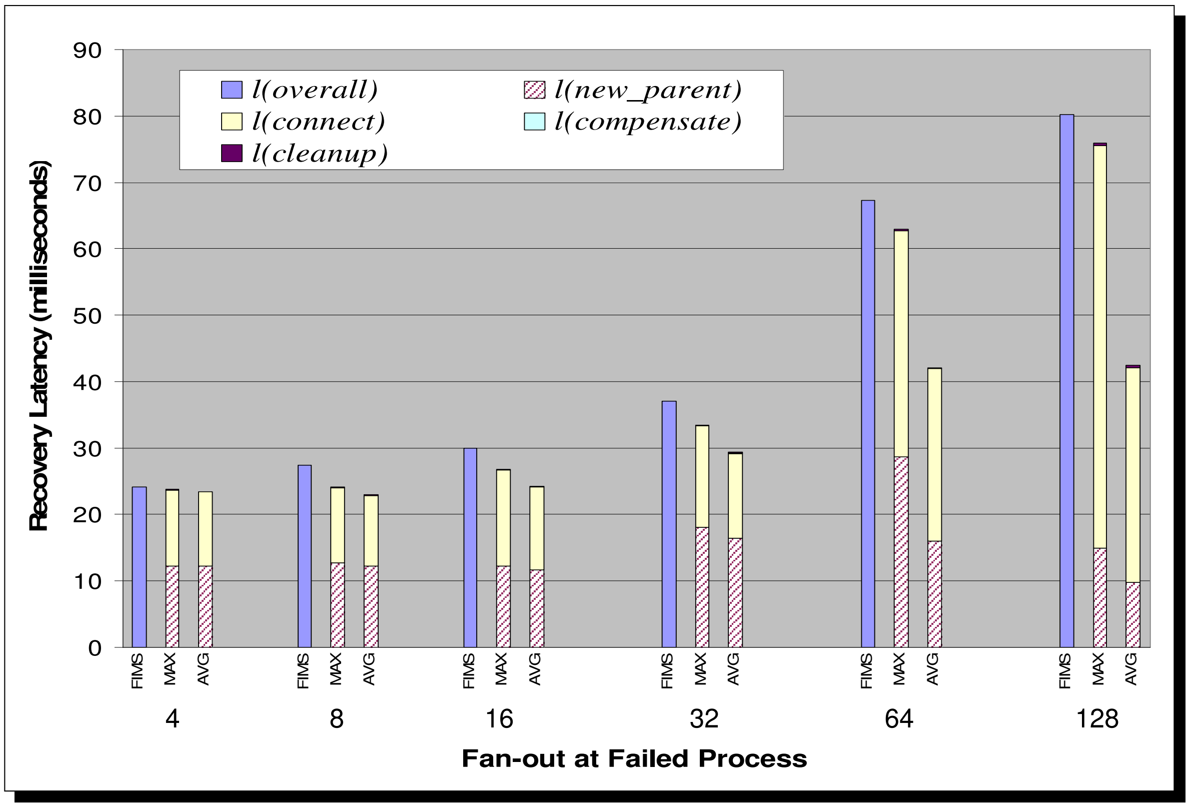


Figure 8.2 Failure Recovery Micro-benchmark Results. $l(overall)$, is the FIMS' conservative overall recovery estimate, $l(new\_parent)$, $l(connect)$, $l(compensate)$ and $l(cleanup)$ are averages of latencies recorded by each orphan to choose a new parent, connect to the new parent, propagate filter state for compensation, and update local data structures.

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, distributing the adoptions over a greater number of adopters would mitigate this contention. For these experiments, $l(new\_parent)$ 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 as we demonstrated in Section 7.3.2, 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 in these results is that even considering FIMS' conservative estimate of overall recovery latency, the latency for our largest fan-out of 128 is less than 80 milliseconds – an insignificant interruption, especially considering that a 128^3 tree has over 2 million leaves.

8.2.4 Application Perturbation Macro-benchmark Results

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 starts with a balanced 2-level process configuration, as shown in Figure 8.1(b), with 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 8.3 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 is injected. Most likely, these are due to other artifacts, like operating system thread scheduling, and we conclude that there is no perceivable change in application performance due to the injected failures. We suspect that if the application data rate were increased to add more stress to the TBŌN, the impact of failures and failure recoveries, may become more noticeable.

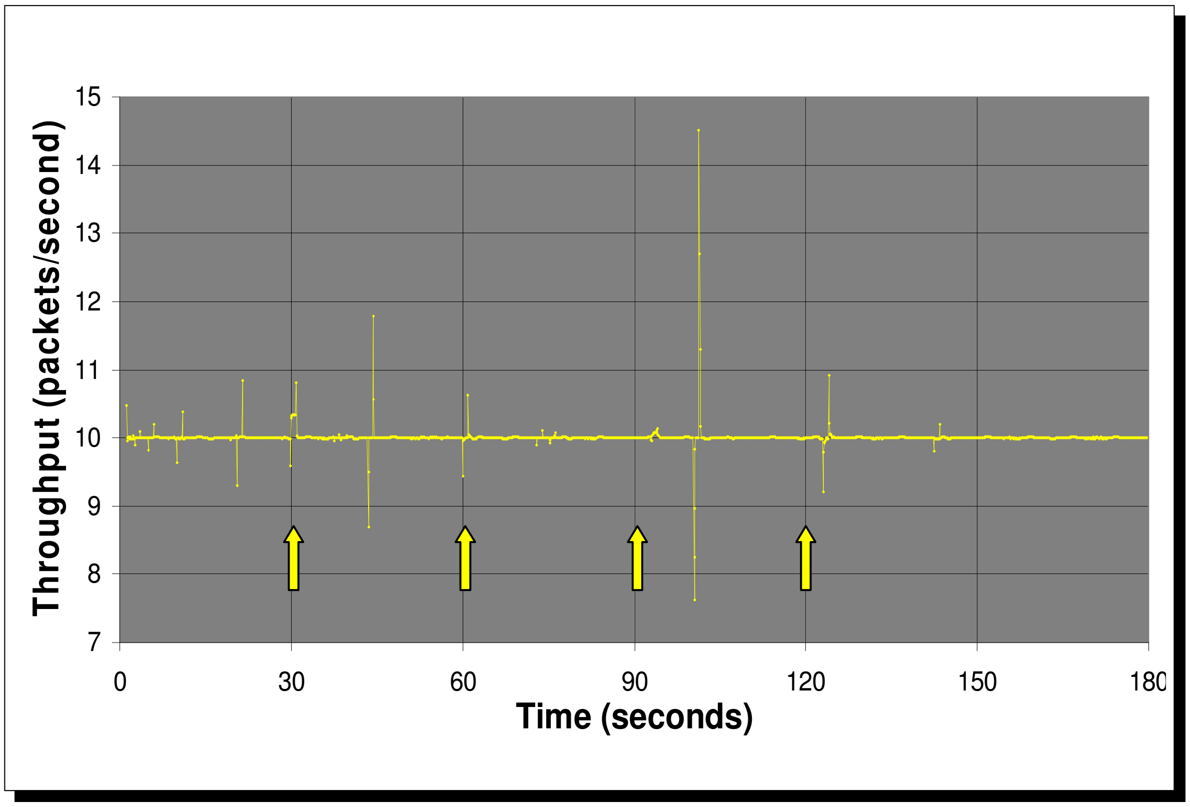


Figure 8.3 Application Perturbation Macro-benchmark Results: A failure (indicated by an arrow) is injected every 30 seconds into a TBŌN initialized with a 32^3 topology.

Chapter 9

Conclusion

Our goal for this research was to develop effective failure recovery mechanisms for high performance TBŌN-based computations in extremely large scale environments. Our fundamental strategy for accomplishing this goal was to avoid the explicit data replication and process coordination mechanisms that prevent existing techniques from operating effectively at large scales. Using this strategy, we developed an efficient and scalable TBŌN failure recovery model. In this final chapter, we summarize the contributions of this work and discuss directions for future work.

9.1 Contributions

In this dissertation, we exploited the characteristics of the TBŌN computational model to enable scalable failure recovery models. The major contributions of this work are:

1. **State Compensation**: Our primary contribution is a novel TBŌN failure recovery model that we call state compensation. State compensation leverages the information redundancies amongst TBŌN process and channel states. Redundant information from the filter state of processes that survive failures are used to compensate for

information lost due to failures. Because state compensation exploits implicit data replication, it requires no additional computational, network or storage resources in the absence of failures. When failures do occur, a small number (O(fan-out)) of TBŌN processes participate in failure recovery, so failure recovery is very scalable.

State compensation requires that data aggregation operations be associative and commutative; these are general properties of our TBŌN computational model and admissive of many computations. Our primary state compensation mechanism, state composition, entails lightweight recovery mechanisms but requires that data aggregation operations be idempotent. The majority of our current MRNet data aggregation operations are idempotent. State decomposition addresses non-idempotent data aggregation operations but requires more heavyweight failure recovery mechanisms.

2. **A Formal Model for Data Aggregation**: We developed a formal specification of our TBŌN-based data aggregation model. This specification allowed us to formalize and validate our state compensation failure recovery mechanisms. Using the formalisms, we identified the requirements of state composition and state decomposition, as well as their constraints and limitations. The formal specification led to a straightforward implementation of state composition and can serve as the basis for exploring extensions of this work including those describe in the next section.

3. **Scalable Tree Reconfiguration**: While previous work in tree reconfiguration algorithms has focused on spanning tree formation and the transmission latency between directly connected processes, we targeted algorithms for applications based on high performance data aggregation. Such applications require high throughput, low latency communication of possibly large amounts of data. Therefore, we focused on:

   (a) the costs of disseminating and managing the TBŌN process information needed by the reconfiguration algorithms,

   (b) the execution times of the reconfiguration algorithms, and

   (c) the data aggregation latency of the resulting configurations.

   For a TBŌN with over one million application back-ends, the most demanding algorithm requires 416 kilobytes of process data and executes in less than 700 milliseconds. We concluded that the primary consideration should be the data aggregation latency of the TBŌN configuration that results from the execution of the tree reconfiguration algorithm. We observed that the least restrictive tree reconfiguration algorithms, in which the majority of the TBŌN processes can adopt orphans, result in low, well-balanced fan-outs. Our final recommendation is that we choose an algorithm that considers all TBŌN processes but restricts increases in tree height; we avoid height increases because they can have a significant negative effect on data aggregation performance.

4. **Scalable, Reliable TBŌN Framework**: We extended the MRNet TBŌN prototype with a complete implementation of the state composition failure recovery model. The extended framework includes failure detection, tree reconfiguration and state compensation mechanisms. Our experiments with this framework show that for TBŌNs that can support millions of application back-end processes, state composition has low (sub-second) failure recovery latencies and inconsequential application perturbation. Researchers and developers now can download the extended MRNet software to develop reliable, scalable tools and applications. Finally, the implementation provides a framework that can be used to implement and evaluate future research extensions.

9.2 Future Research Directions

In Section 6.6, we presented several strategies for extending our failure recovery model, for example, to accommodate heterogeneous filter compositions, application back-end failures, and DAG topologies. We highlight two additional areas that present opportunities for future exploration. The first is the area of scalable, autonomic computing and the second is fault-tolerant TBŌN-based applications.

As HPC system sizes continue to increase, manual system management becomes increasingly prohibitive, and autonomic computing systems will become imperative. Autonomic systems are generic, self monitoring, self (re)configuring, self healing, and self optimizing; such systems operate effectively without human intervention and expertise. This work represents our initial study of self healing, self configuring systems.

As applications are executed on larger numbers of computational nodes, it is important that the applications utilize these nodes as efficiently as possible. An open question is whether we can develop efficient techniques for dynamic, automated TBŌN reconfiguration to optimize application performance and resource utilization. In addition to the reconfiguration mechanisms explored in this work, such TBŌN self optimization entails:

- **TBŌN performance modeling** to determine how application performance is impacted by the TBŌN topology, the data aggregation operations being executed, and the workload being offered to the TBŌN, and
- **self monitoring** to determine dynamically the TBŌN's workload and performance and accurately parametrize our performance models to determine how the process organization can be optimized.

Such performance monitoring and analysis also may determine that resources are being under utilized, in which case we envision mechanisms for removing nodes or processes from the tree and relegating them as spares to replace nodes or processes that fail.

A majority of our early TBŌN data aggregation operations are motivated by the data analysis requirements of parallel and distributed system tools. However, in the early stages of our research, we observed that the TBŌN computational model is useful for computing applications [6], and current work continues to explore additional types of applications and algorithms for which the model is well-suited. 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.

LIST OF REFERENCES

[1] Marcos K. Aguilera, Wei Chen, and Sam Toueg. Heartbeat: A Timeout-free Failure Detector for Quiescent Reliable Communication. *11th International Workshop on Distributed Algorithms (WDA '97)*, pages 126–140, September 1997.

[2] Carlos Almeida and Paulo Verissimo. Timing Failure Detection and Real-time Group Communication in Real-time Systems. *8th Euromicro Workshop on Real-Time Systems*, June 1996.

[3] Peter A. Alsberg and John D. Day. A Principle for Resilient Sharing of Distributed Resources. *2nd International Conference on Software Engineering (ICSE '76)*, pages 562–570, San Francisco, CA, 1976. IEEE Computer Society Press.

[4] Yair Amir and Claudiu Danilov. Reliable Communication in Overlay Networks. *International Conference on Dependable Systems and Networks (DSN03)*, pages 511–520, San Francisco CA, June 2003.

[5] Dorian C. Arnold, Dong H. Ahn, Bronis R. de Supinski, Gregory Lee, Barton P. Miller, and Martin Schulz. Stack Trace Analysis for Large Scale Applications. *21st IEEE International Parallel & Distributed Processing Symposium (IPDPS '07)*, Long Beach, CA, March 2007.

[6] Dorian C. Arnold, Gary D. Pack, and Barton P. Miller. Tree-based Computing for Scalable Applications. *11th International Workshop on High-Level Parallel Programming Models and Supportive Environments*, Rhodes, Greece, April 2006.

[7] B.R. Badrinath and Pradeep Sudame. Gathercast: the Design and Implementation of a Programmable Aggregation Mechanism for the Internet. *9th International Conference on Computer Communications and Networks*, pages 206–213, Las Vegas, NV, October 2000.

[8] Magdalena Balazinska, Hari Balakrishnan, Samuel Madden, and Michael Stonebraker. Fault-tolerance in the Borealis Distributed Stream Processing System. *SIGMOD International Conference on Management of Data*, pages 13–24, Baltimore, MD, June 2005.

[9] Susanne M. Balle, John Bishop, David LaFrance-Linden, and Howard Rifkin. *Applied Parallel Computing*, volume 3732/2006 of *Lecture Notes in Computer Science*, chapter 2, pages 207–216. Springer, February 2006.

[10] Joel F. Bartlett. A NonStop Kernel. *Eighth ACM Symposium on Operating Systems Principles (SOSP '81)*, pages 22–29, Pacific Grove, CA, 1981. ACM.

[11] Jon Louis Bentley and Michael Ian Shamos. Divide-and-conquer in Multidimensional Space. *ACM Symposium on Theory of Computing (STOC '76)*, pages 220–230, Hershey, PA, 1976. ACM.

[12] Bharat Bhargava and Shy-Renn Lian. Independent Checkpointing and Concurrent Rollback for Recovery–An Optimistic Approach. *Symposium on Reliable Distributed Systems*, pages 3–12, 1988.

[13] Andrew D. Birrell, Roy Levin, Michael D. Schroeder, and Roger M. Needham. Grapevine: An Exercise in Distributed Computing. *Communications of the ACM*, 25(4):260–274, 1982.

[14] IBM System Blue Gene Solution. http://www-03.ibm.com/systems/deepcomputing/bluegene/.

[15] Roger Bollo, Jean-Pierre Le Narzul, Michel Raynal, and Frederic Tronel. Probabilistic Analysis of a Group Failure Detection Protocol. *Fourth International Workshop on Object-Oriented Real-Time Dependable Systems*, page 156. IEEE Computer Society, 1999.

[16] Anita Borg, Jim Baumbach, and Sam Glazer. A Message System Supporting Fault Tolerance. *9th ACM Symposium on Operating System Principles*, pages 90–99, Bretton Woods, NH, October 1983.

[17] Daniele Briatico, Augusto Ciufoletti, and Luca Simoncini. A Distributed Domino-effect Free Recovery Algorithm. *4th IEEE Symposium on Reliability in Distributed Software and Database Systems*, pages 207–215, Silver Spring, MD, October 1984.

[18] Michael J. Brim and Barton P. Miller. Group File Operations for Scalable Tools and Middleware. Technical Report UW-CS 1638, University of Wisconsin, 2008.

[19] Bryan Buck and Jeffrey K. Hollingsworth. An API for Runtime Code Patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, Winter 2000.

[20] K. Mani Chandy and Leslie Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[21] Jen-Yeu Chen, Gopal Pandurangan, and Dongyan Xu. Robust Computation of Aggregates in Wireless Sensor Networks: Distributed Randomized Algorithms and Analysis. *4th international Symposium on Information Processing in Sensor Networks (IPSN '05)*, Los Angeles, CA, April 2005. IEEE Press.

[22] Wei Chen, Sam Toueg, and Marcos K. Aguilera. On the Quality of Service of Failure Detectors. *IEEE Transactions on Computers*, 51(5):561–580, May 2002.

[23] Yuqun Chen, Kai Li, and James S. Plank. CLIP: A Checkpointing Tool for Message-passing Parallel Programs. *SuperComputing '97*, San Jose, CA, 1997.

[24] M. Chereque, D. Powell, P. Reynier, J.L. Richier, and J. Voiron. Active Replication in Delta-4. *22nd International Symposium on Fault-Tolerant Computing (FTCS-22)*, pages 28–37, Boston, MA, July 1992.

[25] Dah Ming Chiu, Stephen Hurst, Miriam Kadansky, and Joseph Wesley. TRAM: A Tree-based Reliable Multicast Protocol. Technical Report TR 98-66, Sun Microsystems, July 1998.

[26] Flaviu Cristian and Farnam Jahanian. A Timestamp-Based Checkpointing Protocol for Long-Lived Distributed Computations. *Tenth Symposium on Reliable Distributed Systems*, pages 12–20, Pisa, Italy, September 1991. IEEE Computer Society Press.

[27] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: Towards a Realistic Model of Parallel Computation. *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP '93)*, pages 1–12, San Diego, CA, 1993.

[28] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[29] Pavlos S. Efraimidis and Paul G. Spirakis. Weighted Random Sampling with a Reservoir. *Information Processing Letters*, 97(5):181–185, 2006.

[30] Elmootazbellah N. Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A Survey of Rollback-recovery Protocols in Message-passing Systems. *ACM Computing Surveys*, 34(3):375–408, 2002.

[31] Elmootazbellah N. Elnozahy, David B. Johnson, and Willy Zwaenpoel. The Performance of Consistent Checkpointing. *11th IEEE Symposium on Reliable Distributed Systems*, Houston, TX, 1992.

[32] Elmootazbellah N. Elnozahy and James. 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.

[33] David A. Evensky, Ann C. Gentile, L. Jean Camp, and Robert C. Armstrong. Lilith: Scalable Execution of User Code for Distributed Computing. *6th IEEE International Symposium on High Performance Distributed Computing (HPDC 97)*, pages 306–314, Portland, OR, August 1997.

[34] Christos Faloutsos, H. V. Jagadish, and N. D. Sidiropoulos. Recovering Information from Summary Data. *23rd International Conference on Very Large Data Bases (VLDB '97)*, pages 36–45, Athens, Greece, August 1997. Morgan Kaufmann.

[35] Stuart I. Feldman and Channing B. Brown. IGOR: A System for Program Debugging via Reversible Execution. *1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging (PADD '88)*, pages 112–123, New York, NY, 1988. ACM Press.

[36] Sally Floyd, Van Jacobson, Steve McCanne, Ching-Gung Liu, and Lixia Zhang. A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing. *ACM SIGCOMM Computer Communication Review, Conference on Applications, technologies, architectures, and protocols for computer communication*, 25(4):342–356, October 1995.

[37] Felix C. Gartner. Fundamentals of Fault-tolerant Distributed Computing in Asynchronous Environments. *ACM Computing Surveys*, 31(1):1–26, 1999.

[38] Garth Gibson, Bianca Schroeder, and Joan Digney. Failure Tolerance in Petascale Computers. *CTWatch Quarterly*, 3(4), November 2007.

[39] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. *Data Mining and Knowledge Discovery*, 1(1):29–53, April 1997.

[40] Indranil Gupta, Robert van Renesse, and Kenneth P. Birman. Scalable Fault-Tolerant Aggregation in Large Process Groups. *2001 International Conference on Dependable Systems and Networks (DSN '01)*, pages 433–442, Gøteborg, Sweden, June/July 2001. IEEE Computer Society.

[41] Jean-Michel Helary, Achour Mostefaoui, and Michel Raynal. Preventing Useless Checkpoints in Distributed Computations. *16th Symposium on Reliable Distributed Systems (SRDS '97)*, pages 183–190, Durham, NC, October 1997. IEEE Computer Society.

[42] Jean-Michel Helary, Robert H. B. Netzer, and Michel Raynal. Consistency Issues in Distributed Checkpoints. *IEEE Transactions on Software Engineering*, 25(2):274–281, March–April 1999.

[43] Jeong-Hyon Hwang, Magdalena Balazinska, Alexander Rasin, Ugur Cetintemel, Michael Stonebraker, and Stan Zdonik. High-Availability Algorithms for Distributed Stream Processing. *21st International Conference on Data Engineering (ICDE'05)*, pages 779–790, Tokyo, Japan, April 2005.

[44] Soonwook Hwang and Carl Kesselman. A Generic Failure Detection Service for the Grid. Technical Report ISI-TR-568, Information Sciences Institute, University of Southern CA, Feb 2003.

[45] Kenneth E. Iverson. *A Programming Language*. John Wiley & Sons, Inc., New York, NY, 1962.

[46] John Jannotti, David K. Gifford, Kirk L Johnson, M. Frans Kaashoek, and James W. O'Toole. Overcast: Reliable Multicasting with an Overlay Network. *4th Symposium on Operating Systems Design and Implementation (OSDI 2000)*, San Diego, CA, October 2000.

[47] Hongbo Jiang and Shudong Jin. Scalable and Robust Aggregation Techniques for Extracting Statistical Information in Sensor Networks. *26th IEEE International Conference on Distributed Computing Systems (ICDCS '06)*, page 69, Lisboa, Portugal, July 2006. IEEE Computer Society.

[48] Paul R. Kosinski. A Data Flow Language for Operating Systems Programming. *ACM SIGPLAN Notices*, 8(9):89–94, 1973.

[49] Richard E. Ladner and Michael J. Fischer. Parallel Prefix Computation. *Journal of the ACM*, 27(4):831–838, 1980.

[50] Ten H. Lai and Tao H. Yang. On Distributed Snapshots. *Information Processing Letters*, 25(3):153–158, 1987.

[51] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[52] Gregory L. Lee, Dong H. Ahn, Dorian C. Arnold, Bronis R. de Supinski, Matthew Legendre, Barton P. Miller, Martin Schulz, and Ben Liblit. Lessons Learned at 208K: Towards Debugging Millions of Cores. *Supercomputing 2008 (SC2008)*, Austin, TX, November 2008. To Appear.

[53] Juan Leon, Allan L. Fisher, and Peter Steenkiste. Fail-Safe PVM: A Portable Package for Distributed Programming with Transparent Recovery. Technical Report CMU-CS-93-124, Carnegie Mellon University, Pittsburgh, PA, February 1993.

[54] Kai Li, Jeffrey F. Naughton, and James S. Plank. Low-Latency, Concurrent Checkpointing for Parallel Programs. *IEEE Transactions on Parallel and Distributed Systems*, 5(8):874–879, August 1994.

[55] Kai Li, Jeffrfey F. Naughton, and James S. Plank. Real-time, Concurrent Checkpoint for Parallel Programs. *2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP '90)*, pages 79–88, Seattle, Washington, 1990. ACM.

[56] Etnus LLC. TotalView User's Guide, Document version 6.0.0-1, January 2003.

[57] Los Alamos National Laboratory. Operational Data to Support and Enable Computer Science Research. http://institute.lanl.gov/data/fdata/.

[58] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. TAG: a Tiny AGgregation Service for Ad-Hoc Sensor Networks. *5th Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, December 2002.

[59] Amit Manjhi, Suman Nath, and Phillip B. Gibbons. Tributaries and Deltas: Efficient and Robust Aggregation in Sensor Network Streams. *ACM SIGMOD International Conference on Management of Data (SIGMOD 2005)*, pages 287–298, Baltimore, MD, June 2005. ACM Press New York, NY, USA.

[60] Friedemann Mattern. Virtual Time and Global States of Distributed Systems. *International Workshop on Parallel and Distributed Algorithms*, pages 215–226, Chateau De Bonas, Gers, France, 1989. North-Holland.

[61] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*, July 1997.

[62] Alberto Montresor, Mark Jelasity, and Ozalp Babaoglu. Robust Aggregation Protocols for Large-Scale Overlay Networks. *2004 International Conference on Dependable Systems and Networks (DSN 2004)*, page 19, Palazzo dei Congressi, Florence, Italy, June/July 2004. IEEE Computer Society.

[63] Aroon Nataraj, Allen D. Malony, Alan Morris, Dorian Arnold, and Barton Miller. A Framework for Scalable, Parallel Performance Monitoring using TAU and MRNet. *International Workshop on Scalable Tools for High-End Computing (STHEC 2008)*, Island of Kos, Greece, June 2008.

[64] Suman Nath, Phillip B. Gibbons, Srinivasan Seshan, and Zachary Anderson. Synopsis Diffusion for Robust Aggregation in Sensor Networks. *ACM Transactions on Sensor Networks*, 4(2):1–40, 2008.

[65] National Energy Research Scientific Computing Center. FY07 System Availability Statistics. http://www.nersc.gov/nusers/status/AvailStats/FY07/.

[66] Robert H. B. Netzer and Jian Xu. Necessary and Sufficient Conditions for Consistent Global Snapshots. *IEEE Transactions on Parallel and Distributed Systems*, 6(2):165–169, February 1995.

[67] Katia Obraczka. Multicast Transport Protocols: A Survey and Taxonomy. *Communications Magazine, IEEE*, 36(1):94–102, January 1998.

[68] Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh. The Design and Implementation of Zap: A System for Migrating Computing Environments. *5th Symposium on Operating Systems Design and Implementation (OSDI 2002)*, pages 361–376, Boston, MA, December 2002.

[69] Douglas Z. Pan and Mark A. Linton. Supporting Reverse Execution for Parallel Programs. *1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging (PADD '88)*, pages 124–129, Madison, WI, 1988. ACM Press.

[70] Fernando Pedone and Svend Frølund. Pronto: A Fast Failover Protocol for Off-the-shelf Commercial Databases. *19th IEEE Symposium on Reliable Distributed Systems (SRDS '00)*, pages 176–185, Nürnberg, Germany, October 2000. IEE Computer Society.

[71] Dimitrios Pendarakis, Sherlia Shi, Dinesh Verma, and Marcel Waldvogel. ALMI: An Application Level Multicast Infrastructure. *3rd USNIX Symposium on Internet Technologies and Systems (USITS '01)*, San Francisco, CA, March 2001.

[72] James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li. Libckpt: Transparent Checkpointing under Unix. *USENIX Winter 1995 Technical Conference*, pages 213–224, New Orleans, LA, January 1995.

[73] Michael L. Powell and David L. Presotto. PUBLISHING: A Reliable Broadcast Communication Mechanism. *9th ACM Symposium on Operating System Principles*, pages 100–109, Bretton Woods, NH, October 1983.

[74] Brian Randell. System Structure for Software Fault Tolerance. *International Conference on Reliable Software*, pages 437–449, Los Angeles, CA, 1975.

[75] Robbert Van Renesse, Kenneth P. Birman, and Werner 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.

[76] Phillip C. Roth, Dorian C. Arnold, and Barton P. Miller. MRNet: A Software-Based Multicast/Reduction Network for Scalable Tools. *2003 ACM/IEEE conference on Supercomputing (SC '03)*, page 21, Phoenix, AZ, November 2003. IEEE Computer Society.

[77] Phillip C. Roth and Barton P. Miller. 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.

[78] David L. Russell. State Restoration in Systems of Communicating Processes. *IEEE Transactions on Software Engineering*, 6(2):183–194, 1980.

[79] Federico D. Sacerdoti, Mason J. Katz, Matthew L. Massie, and David E. Culler. Wide Area Cluster Monitoring with Ganglia. *IEEE International Conference on Cluster Computing (CLUSTER 2003)*, pages 289–298, Hong Kong, September 2003.

[80] J. Saia and A. Trehan. Picking up the Pieces: Self-Healing in Reconfigurable Networks. *22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS '08)*, Miami, FL, April 2008.

[81] Fred B. Schneider. Implementing Fault-tolerant Services using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.

[82] Bianca Schroeder and Garth Gibson. The Computer Failure Data Repository. *Workshop on Reliability Analysis of System Failure Data (RAF'07)*, Cambridge, UK, March 2007.

[83] Bianca Schroeder and Garth A. Gibson. A Large-scale Study of Failures in High-performance Computing Systems. *Dependable Systems and Networks (DSN 2006)*, Philadelphia, PA, June 2006.

[84] Bianca Schroeder and Garth A. Gibson. Understanding Failures in Petascale Computers. *Journal of Physics Conference Series*, 78(1), 2007.

[85] Mark Seager. Operational Machines: ASCI White. *7th Workshop on Distributed Supercomputing*, Durango, CO, March 2003. Presentation.

[86] Luis Moura Silva. *Checkpointing mechanisms for Scientific Parallel Applications*. PhD thesis, University of Coimbra, Portugal, March 1997.

[87] Matthew J. Sottile and Ronald G. Minnich. Supermon: A High-speed Cluster Monitoring System. *IEEE International Conference on Cluster Computing (CLUSTER 2002)*, pages 39–46, Chicago, IL, September 2002.

[88] Fred Stann and John Heidemann. RMST: reliable data transport in sensor networks. *2003 IEEE International Workshop on Sensor Network Protocols and Applications*, pages 102–112, May 2003.

[89] Paul Stelling, Ian Foster, Carl Kesselman, Craig Lee, and Gregor von Laszewski. A Fault Detection Service for Wide Area Distributed Computations. *7th IEEE International Symposium on High Performance Distributed Computing*, pages 268–278, Chicago, IL, July 1998.

[90] Georg Stellner. CoCheck: Checkpointing and Process Migration for MPI. *International Parallel Processing Symposium*, pages 526–531, Honolulu, HI, April 1996. IEEE Computer Society.

[91] Erich Strohmaier, Jack J. Dongarra, Hans W. Meuerd, and Horst D. Simone. Recent Trends in the Marketplace of High Performance Computing. *Parallel Computing*, 31(3–4):261–273, March–April 2005.

[92] Rob Strom and Shaula Yemini. Optimistic Recovery in Distributed Systems. *ACM Transactions on Computer Systems*, 3(3):204–226, 1985.

[93] Yuval Tamir and Carlo H. Sequin. Error Recovery in Multicomputers Using Global Checkpoints. *13th International Conference on Parallel Processing*, pages 32–41, Bellaire, MI, August 1984.

[94] The BlueGene/L Team. An Overview of the BlueGene/L Supercomputer. *2002 ACM/IEEE conference on Supercomputing (Supercomputing '02)*, pages 1–22. IEEE Computer Society Press, 2002.

[95] Thinking Machines Corporation. Prism User's Guide, December 1991.

[96] Zhijun Tong, Richard Y. Kain, and W. T. Tsai. Rollback Recovery in Distributed Systems Using Loosely Synchronized Clocks. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):246–251, 1992.

[97] Top 500 Supercomputer Sites. http://www.top500.org/ (visited February 2007).

[98] Chieh-Yih Wan, Andrew T. Campbell, and Lakshman Krishnamurthy. Pump-Slowly, Fetch-Quickly (PSFQ): A Reliable Transport Protocol for Sensor Networks. *IEEE Journal on Selected Areas in Communications*, 23(4):862–872, April 2005.

[99] Yi-min Wang. Reducing Message Logging Overhead for Log-based Recovery. *IEEE International Symposium on Circuits and Systems (ISCAS '93)*, volume 3, pages 1925–1928, Chicago, IL, May 1993.

[100] Yi-Min Wang. Consistent Global Checkpoints that Contain a Given Set of Local Checkpoints. *IEEE Transactions on Computers*, 46(4):456–468, 1997.

[101] Fetahi Wuhib, Mads Dam, Rolf Stadler, and Alexander Clemm. Robust Monitoring of Network-wide Aggregates through Gossiping. *10th IFIP/IEEE International Symposium on Integrated Network Management. (IM '07)*, pages 226–235, 2007.

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

[103] Mengkun Yang and Zongming Fei. A Proactive Approach to Reconstructing Overlay Multicast Trees. *INFOCOM 2004*, Hong Kong, March 2004.

[104] Yong Yao and J. E. Gehrke. Query Processing in Sensor Networks. *First Biennial Conference on Innovative Data Systems Research (CIDR 2003)*, Asilomar, CA, January 2003.

[105] Victor C. Zandy and Barton P. Miller. Reliable Network Connections. *ACM MobiCom*, Atlanta, GA, September 2002.

[106] Victor C. Zandy, Barton P. Miller, and Miron Livny. Process Hijacking. *8th International Symposium on High Performance Distributed Computing (HPDC '99)*, pages 177–184, Redondo Beach, CA, August 1999.

[107] Beichuan Zhang, S. Jamin, and Lixia Zhang. Host Multicast: A Framework for Delivering Multicast to End Users. *21st Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2002)*, pages 1366–1375, June 2002.

[108] Yuanyuan Zhou, Peter M. Chen, and Kai Li. Fast Cluster Failover using Virtual Memory-mapped Communication. *13th International Conference on Supercomputing (ICS '99)*, pages 373–382, Rhodes, Greece, 1999. ACM.