Handling Cascading Failures: The Case for Topology-Aware Fault-Tolerance *

Soila Pertet and Priya Narasimhan
Electrical & Computer Engineering Department
Carnegie Mellon University
5000 Forbes Avenue, Pittsburgh, PA 15213-3890
spertet@ece.cmu.edu, priya@cs.cmu.edu

Abstract

Large distributed systems contain multiple components that can interact in sometimes unforeseen and complicated ways; this emergent "vulnerability of complexity" increases the likelihood of cascading failures that might result in widespread disruption. Our research explores whether we can exploit the knowledge of the system's topology, the application's interconnections and the application's normal fault-free behavior to build proactive fault-tolerance techniques that could curb the spread of cascading failures and enable faster system-wide recovery. We seek to characterize what the topology knowledge would entail, quantify the benefits of our approach and understand the associated tradeoffs.

1. Introduction

Cascading failures occur when local disturbances ripple through interconnected components in a distributed system, causing widespread disruption. For example, in 1990, a bug in the failure recovery code of the AT&T switches [10] led to cascading crashes in 114 switching nodes, 9 hours of downtime and at least $60 million in lost revenue. A more recent example of cascading failure is the electric power blackout [15] of August 2003 that spread through the Midwest and Northeast U.S. and part of Canada, affecting 50 million people and costing $4-10 billion! Other examples include the escalation of a divide-by-zero exception into a Navy ship's network failure and subsequent grounding [4], and cascading failures in the Internet [8, 9]. Software upgrades, although they are not considered to be faults, can manifest themselves as cascading failures; this is because an upgrade in one part of the system can trigger dependent upgrades (with accompanying performance degradation and outages [6]) in other parts of the system.

* This work is partially supported by the NSF CAREER Award CCR-0238381, Bosch Research, the DARPA PCES contract F33615-03-C-4110, and the General Motors Collaborative Laboratory at CMU.

Clearly, cascading failures involve significant downtime with financial consequences. Given the kinds of large, interconnected distributed applications that we increasingly deploy, system developers should consider the impact of cascading failures, and mitigation strategies to address them. The classical approach to dealing with cascading failures is fault-avoidance where systems are partitioned so that the behavior and performance of components in one sub-system is unaffected by components in another subsystem. Transactional systems avoid cascading aborts through atomic actions, for instance, by ensuring that no reads occur before a write commits. Unfortunately, fault-avoidance does not always work – systems are often put together from Commercial Off-The-Shelf (COTS) components or developed hastily to meet time-to-market pressure, leading to complex systems with dependencies and emergent interactions that are not very well understood and with residual faults that manifest themselves during operation. Transactional models can also be too expensive to implement in some distributed systems due to the latency in ensuring atomicity.

Our research focuses on ways of building distributed systems capable of detecting and recovering from cascading failures in a timely and cost-effective manner. This research is novel because most dependable systems tend to deal with independent, isolated failures; furthermore, they do not exploit knowledge of system topology or dependencies for more effective, directed recovery. We seek to understand if *topology-aware fault-tolerance* can be effective, i.e., can we exploit knowledge of system topology to contain cascading failures and to reduce fault-recovery latencies in distributed systems? Here, we define topology as the static dependencies between an application's components, along with the dynamic dependencies that arise when the application's components are instantiated onto physical resources. We also seek to discover whether certain topologies are more resilient to cascading failures than others are; this could potentially provide system developers with "rules-of-thumb" for structuring systems to tolerate such failures.

The paper is organized as follows: Section 2 introduces our topology-aware fault-tolerance approach. Section 3 dis-

Failure Class	Description	Examples
Malicious faults	Deliberate attacks that exploit system vulnerabilities to propagate from one system to the next	Viruses and worms
Recovery Escalation	Failures during execution of recovery routines lead to subsequent failures in neighboring nodes	AT&T switch failure [10], cascading abort in transactional systems
Unhandled exceptions	Unanticipated failure conditions result in cascading exceptions	Unhandled system exception
Cumulative, progressive error propagation	Severity of the failure increases in magnitude as failure propagates from node-to-node.	Cascading timeouts, value-fault propagation (Ariane5)
Chained reconfiguration	Transient errors leading to slew of reconfiguration	Routing instability in BGP, live upgrades in distributed systems

Table 1. A taxonomy of cascading failures

cusses our proposed framework. Section 4 presents a preliminary case study which motivates the use of topology-aware fault-tolerance approaches. Section 5 critiques our current system and discusses our future directions. Section 6 discusses related work and Section 7 concludes.

2. Our Approach in a Nutshell

Topology-aware fault-tolerance aims to exploit knowledge of the system's topology and the application's normal fault-free behavior to build proactive fault-tolerance techniques that curb the spread of cascading failures and enable faster system-wide recovery. For instance, if we provided each component in the system with some knowledge of other components in its dependency path (beyond its immediate neighbors), could this knowledge enhance the component's resilience to cascading failures? We outline below some of the key research questions that we are hoping to address in this research. While our preliminary results will only address a sub-set of these questions, through our participation in the *HotDep Workshop*, we hope to receive feedback on our preliminary results and the potential of our approach to influence dependable systems development.

2.1. Key Research Questions

Classification of cascading failures: We need to understand the classes of cascading failures that can occur in distributed systems. We need to explore if there are any similarities between the classes of failures, and if so, whether we can develop a generic approach to curb the spread of failures and ensure faster system-wide recovery. Table 1 illustrates our initial taxonomy of cascading failures.

Dependency characterization: Dependencies in distributed systems arise due to the invocation (or static-call) graph and the dynamic mapping of application processes to resources. We need to determine the "strength" of these dependencies, i.e., the likelihood that a component is affected if its "parent" fails. Another question is how much information about the dependency graph a component would need in order to become more resilient to cascading failures, e.g., should the component be aware only of other components that are within 2 hops from it, or does it need to know the entire dependency graph? How does the underlying system topology influence the spread of cascading failures, and are some topologies more resilient to cascading failures than others?

Failure-detection: What information, if any, would we need to disseminate through the system to detect/diagnose cascading failures? How accurate should this information be? Also, what is the consequence of disseminating inaccurate information through the system?

Fault-recovery: How effective are existing fault-recovery strategies in curbing cascading failures? How can we coordinate recovery across multiple distributed components in a way that preserves system consistency? Can we develop bounds for topology-aware fault-recovery latencies?

Evaluation: How do we inject cascading faults in large distributed systems for the purpose of evaluation? What metrics should we use to evaluate the effectiveness of our approach? For instance, would response times and throughput be sufficient, or would the metric depend on the class of cascading failure that we aim to tolerate?

3. Proposed Framework

We envision a system where each node is equipped with a middleware layer that monitors the "health" of the node based on local and external data. (see Figure 1). Local data supplies system-level and application-level metrics specific to that node, $e.g.$, workload, resource usage and error logs. External data supplies metrics from other nodes on the dependency path. When the middleware layer suspects that a problem has occurred at a node, it first tries to pinpoint the cause of the failure using local data. However, for cascad-

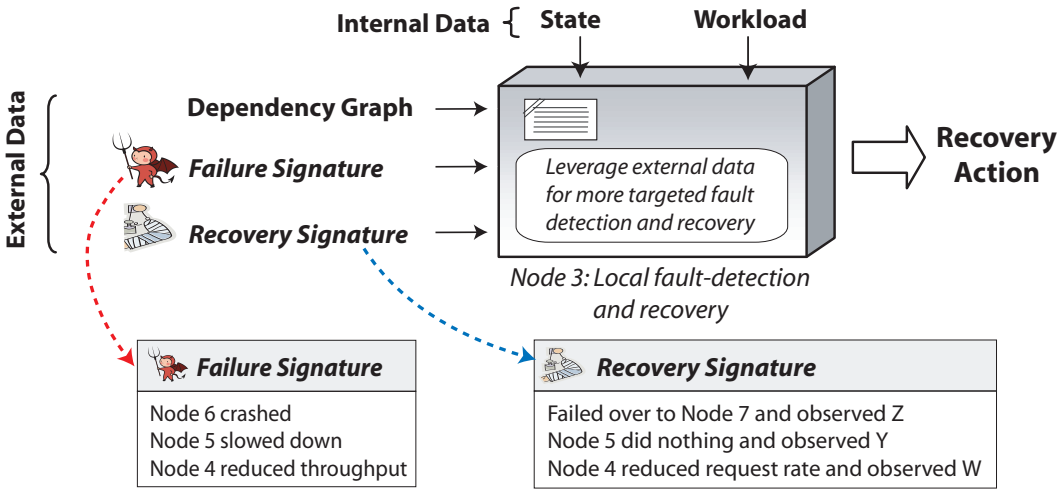

Figure 1. Framework for topology-aware fault-tolerance.

ing failures, the local data might not be sufficient to diagnose the problem since the source of the failure might be several nodes away (see Section 4). In this case, the node leverages external data to gain a more global perspective of the system for better fault-diagnosis and directed fault-recovery. External data consists of dependency graphs, failure signatures and fault-recovery signatures.

Dependency Graph: This graph represents the dependencies between nodes, and the "strength" of these dependencies. Dependencies may arise due to shared resources and/or shared messages. Several approaches have been proposed to track dependencies in distributed systems. We classified these approaches into three categories namely: fault-free/normal dependencies, failure-driven dependencies and recovery-driven dependencies (see Figure 2).

- *Fault-free/normal dependencies:* The dependency graph typically captures dependencies that arise during the fault-free invocation of components in a system. The "strength" of the dependency in this graph may be characterized by the frequency with which a component is invoked. For example, Sailer et al [1] exploit application-level dependencies for problem determination in e-commerce systems.

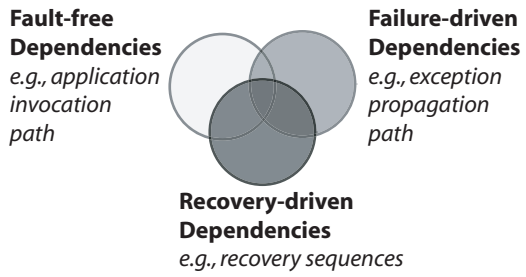

Figure 2. System Dependencies.

- *Failure-driven dependencies:* The dependency graph captures the error propagation paths in a system. For example, Candea et al [3] use reflective middleware to discover exception propagation paths in componentized Internet applications. Jhumka et al [7] use a white-box approach to compute the probability that an input error will propagate from the source node to the target node. These probabilities serve as weights on their dependency graph.
- *Recovery-driven dependencies:* These dependencies arise due to the need to properly sequence fault recovery actions so that interdependency conditions are met, $e.g.$, resource A can be recovered only after resource B is up and running. Buskens et al [13] present a dependable model that properly sequences recovery actions across components to achieve faster system initialization.

The approach adopted for characterizing dependencies depends on the failure model used. For example, tracking fault-free/normal dependencies might be more appropriate for diagnosing performance slowdowns, whereas tracking failure-driven dependencies might be more appropriate when dealing with application exceptions. An interesting question would be to determine which approach provides the largest coverage for diagnosing failures.

Failure signatures: A node also receives information on failures which have occurred in other nodes along its dependency graph, $e.g.$, *node 6 has crashed*. Nodes might also share their suspicions on failures, $e.g.$, *node 4 thinks node 6 has crashed*, and warnings based on resource usage $e.g.$, *node 5 has unusually high CPU utilization* . These failure signatures might improve the chances correctly assigning blame to nodes and localizing the root cause of failures to a list of likely suspects. In addition to assigning blame, the failure signatures could also allow nodes to declare their in-

nocence. This would eliminate them from the list of suspects, and furthur drill down on the cause of the failure. We would need a mechanism to ensure that the information in the failure signatures is trustworthy.

Fault recovery: Once the problem has been diagnosed, a node can then take the appropriate fault-recovery action and propagate information on the recovery action it has taken to other nodes along its dependency path. The node also uses recovery-driven dependencies to properly sequence its recovery actions so that it does not take recovery actions that conflict with the recovery actions taken by other nodes.

3.1. Coping with Complexity

Nodes in large distributed systems currently maintain a great deal of system-level and application-level metrics. Topology-aware fault-tolerance appears to aggravate the problem by requiring that nodes maintain information about other nodes along their dependency path (beyond their immediate neighbors)! Part of the problem with today's systems is that system administrators can be overloaded with metrics and may experience great difficulty sifting out the relevant metrics to diagnose the problem. We believe that the strength of a topology-aware fault-tolerance approach would be to quantify the influence of failures in one node on other nodes in the system, for certain classes of failures. This information could highlight problem areas in a system and provide insight on how distributed systems can be built from scratch to be intrinsically robust in the face of complex, cascading failures.

4. Case Study: Overload in Sensor Network

This case study outlines a scenario where topology-aware fault-tolerance can be applied. We consider a simple sensor network [12] consisting of a collection of sensors, intermediate hubs and a centralized smart hub. The sensors collect environmental data and periodically push this data to the smart hub via the intermediate hubs. Hubs can locate other hubs through a naming service, while sensors can only communicate with the hubs closest to them due to their limited communication range. We consider the case where a sensor malfunctions and floods the network with meaningless data. Ideally, the hub nearest to the faulty sensor should detect this failure and shut this sensor down. If this hub does not have sufficient time, resources or intelligence to contain the failure, other nodes in the system will experience network congestion. The smart hub might be most adversely affected by the failure because most of the system traffic is aggregated at this hub.

There are two approaches that the system could take to isolate the faulty sensor namely: (i) traversing from hub-to-hub to identify the path whose traffic deviates the most from the norm – *(we refer to this as our baseline)*; or (ii) the hub might know enough of the system topology, beyond its immediate neighbors, to pinpoint the sensor with the highest deviation from the norm and shut it down directly. We refer to this scheme as our *topology-aware scheme*. This study represents our first attempt at characterizing merely one kind of distributed applications that would benefit from topology-aware fault-recovery.

4.1. Test Application

We modeled the sensor network as a distributed multi-staged CORBA [11] application.[1] We implemented the sensors as CORBA clients. The sensors send asynchronous CORBA *oneway* requests every 10ms to the smart hub via an arbitrary number of intermediate hubs. These *oneway* messages are best-effort requests that can be silently dropped if network congestion or other resource shortages occur. We designate the client as stage 0 and servers as stages *1*,..., *N*, where stage N is the maximum number of servers in the chain. We derive the system topology for our topology-aware approach by prepending the dependency information to the CORBA request as it passes from stage-to-stage, for instance, a message from sensor1 may indicate that it passed through hub2 and hub3 before reaching hub4.

We inject faults by periodically inducing a faulty sensor to send a burst of requests that congest the entire network. The smart hub detects a failure when 15 or more consective request latencies exceed a predetermined threshold, and uses the dependency information it has stored to locate the sensor whose traffic deviates the most from the norm. The smart hub then triggers a "reconfiguration" (which will, of course, be specific to the fault source and manifestation; in our case, this is the smart hub sending a synchronous message that causes the faulty sensor to reduce its request rate back to one request every 10ms).

4.2. Preliminary Results

We ran our experiments on seven Emulab [17] nodes with the following specifications: 850MHz processor, 512MB RAM, and TimeSys Linux/NET 3.1 running RedHat Linux 9. We implemented the CORBA application over the TAO ORB (ACE and TAO version 5.4). We used NTP to keep the clocks on the nodes synchronized to 1 ms of each other. Each experiment consisted of 2000 client invocations during which we injected a fault about every 150 invocations.

Comparison of fault-recovery schemes. We measured the following parameters for both the baseline and topology-aware schemes see Table 2:

1 CORBA is a set of standard specifications designed to support platform and language-independent, object-oriented distributed computing.

- *Fault-free request latency*, $i.e.$, the amount of time it takes for a *oneway* request from a sensor to reach the smart hub under fault-free conditions;
- *Reconfiguration latency*, $i.e.$, the amount of time between the detection of the failure and the reconfiguration of the faulty sensor;
- *Reconfiguration jitter*, $i.e.$ the standard deviation (σ) in the reconfiguration latencies;
- *Overhead of tracking topology information* as the percentage increase in fault-free request latencies and the number of additional bytes sent per message;

We found that the reconfiguration latency and jitter in our baseline application became very significant as we increased the number of stages in the system. This is because the reconfiguration process involved communication with a larger set of hubs thereby increasing the data traffic in an already congested system! The average latencies for the baseline scheme increased almost exponentially as we increased the number of stages in the application (see Figure 3). However, the average reconfiguration latencies for the topology-aware scheme increased linearly because we limited the reconfiguration notification to a smaller, and relevant, set of nodes. The overhead introduced by topology tracking was minimal. However, sample application was trivial and furthur investigation of the performance overhead introduced in larger systems is needed to obtain conclusive results.

One possible approach for building a scalable system is to limit the tracking/storage of topology information to critical components, as opposed to storing information for every component in the system. The amount of information that a node maintains also increases with the number of stages. For a large number of stages, we might decompose the system into segments, and apply the topology-aware fault-tolerance routines on an segment-wise basis. For instance, a 10-stage application can be structured into two segments of 5 stages each so that topology-aware reconfiguration notifications need not traverse more than 5 stages at a time – each node in a segment would know about all the nodes in its segment, as well as the node at the edge of its neighboring segments.

STAGES	**2**	**3**	**4**	**5**
Avg. Fault free Latency *(ms)*	0.42	1.03	1.66	2.25
Top. Overhead *(% of latency)*	+6%	+3%	+2%	+2%
Top. Overhead *(bytes/message)*	25	50	75	100
Reconfig. Jitter baseline (σ) *(ms)*	1.38	5.21	***80.24***	***103.3***
Reconfig. Jitter topology (σ) *(ms)*	1.42	1.70	2.08	2.22

Table 2. Summary of Experiment

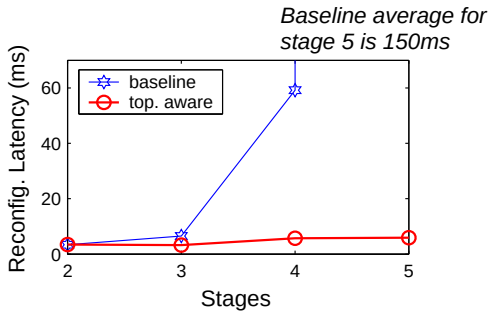

Figure 3. Reconfiguration Latencies.

5. Future Work

To evaluate the effectiveness of our approach, we intend to apply our approach to real-world distributed benchmarks, such as TPC-W [14]. We would also like to explore and experimentally evaluate strategies to handle other kinds of cascading failures, e.g., timeouts, propagating resource exhaustion, live upgrades. In this paper, we have investigated but one fault-recovery strategy – we are yet to implement and compare the effectiveness of other recovery schemes. We also aim to cope with multi-dimensional failures, e.g., what if subsequent failures occur during recovery from a cascading failure? Most of all, in the long term, we believe that our contribution will be greatest if we are able to extract higher-level insights into how systems can be designed and implemented so that they are "born" resistant to cascading failures of various kinds.

6. Related Work

Issarny et al [5] discuss a forward error-recovery approach that uses coordinated atomic actions (CAA) to handle concurrent exceptions in composite web services. The participants of a CAA are aware of their immediate callers and callees. Our system, on the other hand investigates fault-recovery strategies where nodes exploit knowledge of the dependency graph (beyond their immediate neighbors) to achieve faster recovery.

The routing protocol community has also conducted extensive research in containing cascading failures. They center on developing scaleable routing protocols where the extent of disruption caused by a fault depends on the severity of the fault rather than the size of the system. Arora and Zhang [2] present a protocol for local stabilization in shortest path routing (LSRP), which uses diffusing waves to contain fault propagation. Our research, on the other

hand, focuses on how we can transparently apply similar notions of fault-containment to middleware environments like CORBA. These environments present unique challenges since unlike routing protocols, which typically have a limited set of variables and algorithms, middleware environments can host large numbers of processes running diverse software algorithms.

The MAFTIA [16] middleware architecture uses topology-awareness to enhance security through the separation of concerns, for instance, the separation of communication from processing. MAFTIA selects a topology at design-time that enhances certain system properties. In our system, we learn about the underlying topology at run-time based on the application-level data flow, and use this information to send early fault warnings to nodes.

7. Conclusion

This paper highlights the need for components in distributed systems to have some awareness of the system topology, specifically to tolerate cascading failures. We outline the research questions worth addressing in our topology-aware fault-tolerance approach. Through a case study on overload in a sensor network, we highlight a situation where making decisions solely on the basis of local information can aggravate a cascading failure, and show that topology information can provide for targeted, faster recovery. In future work, we hope to quantify the tradeoffs between the expected benefit of better fault detection/containment and the additional complexity of topology-tracking. Our preliminary results suggest some promise for our topology-aware approach, particularly for large distributed systems where cascading failures ought to be considered. Our ultimate aim is to understand how distributed systems (i) can be built from scratch to be intrinsically robust in the face of complex, cascading failures, and (ii) can efficiently maintain, and readily exploit, topology information for many purposes, including faster reconfiguration.

References

[1] M. K. Agarwal, A. Neogi, K. Appleby, J. Faik, G. Kar, and A. Sailer. Threshold management for problem determination in transaction based e-commerce systems. In *To appear in the International Symposium on Integrated Network Management*, Nice, France, May 2005.

[2] A. Arora and H. Zhang. Local stabilization in shortest path routing. In *International Conference on Dependable Systems and Networks*, pages 139–148, San Francisco, CA, June 2003.

[3] G. Candea, M. Delgado, M. Chen, and A. Fox. Automatic failure-path inference: A generic introspection technique for Internet applications. In *IEEE Workshop on Internet Applications*, pages 132–141, San Jose, CA, June 2003.

[4] Gregory Slabodkin. Software glitches leave Navy smart ship dead in the water. Government Computer News, July 1998.

[5] V. Issarny, F. Tartanoglu, A. Romanovsky, and N. Levy. Coordinated forward error recovery for composite Web services. In *International Symposium on Reliable Distributed Systems*, pages 6–18, Florence, Italy, October 2003.

[6] James Cope. Network outage hits AT&T's ATM users. ComputerWorld, Feb. 2001.

[7] A. Jhumka, M. Hiller, and N. Suri. Assessing inter-modular error propagation in distributed software. In *Symposium on Reliable Distributed Systems*, pages 152–161, New Orleans, LA, October 2001.

[8] Mike Martin. Cascading failures could crash the global Internet. NewsFactor Network, Feb. 2003.

[9] A. E. Motter and Y.-C. Lai. Cascade-based attacks on complex networks. *Physical Review E*, 66:065102–1–065102–4, 2002.

[10] P. G. Neumann. Cause of AT&T network failure. *Risks Digest*, 9(62), January 1990.

[11] Object Management Group. The CORBA/IIOP Specification Version 3.0.2. OMG Technical Committee Document formal/2002-12-02, December 2002.

[12] S. Ravula, B. Petrus, J. E. Kim, and C. Stoermer. Position paper: Quality attributes in wireless sensor networks. In *To appear in the Workshop on Software Technologies for Future Embedded and Ubiquituous Systems*, Seattle, WA, May 2005.

[13] Y. Ren, R. Buskens, and O. Gonzalez. Dependable initialization of large-scale distributed software. In *International Conference on Dependable Systems and Networks*, pages 335–344, Florence, Italy, June 2004.

[14] Transaction Processing Performance Council. Tpc benchmark w (web commerce) specification Version 1.8. February 2002.

[15] U.S.-Canada Power System Outage Task Force. Final report on the August 14th blackout in the United States and Canada. April 2004.

[16] P. Veríssimo, N. F. Neves, and M. Correia. The middleware architecture of MAFTIA: A blueprint. In *Proceedings of the IEEE Third Information Survivability Workshop (ISW-2000)*, Boston, MA, October 2000.

[17] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, December 2002.