

# FAULT TOLERANT DISTRIBUTED INFORMATION SYSTEMS

John C. Knight Matthew C. Elder

Department of Computer Science, University of Virginia  
151 Engineer's Way  
PO Box 400740  
Charlottesville, VA 22904-4740  
USA

(knight | mce7e)@cs.virginia.edu

+1 804 982 2216 (Voice)

+1 804 982 2214 (FAX)

**Abstract.** Critical infrastructures provide services upon which society depends heavily; these applications are themselves dependent on distributed information systems for all aspects of their operation and so survivability of the information systems is an important issue. Fault tolerance is a key mechanism by which survivability can be achieved in these information systems. We outline a specification-based approach to fault tolerance, called RAPTOR, that enables systematic structuring of fault tolerance specifications and an implementation partially synthesized from the formal specification. The RAPTOR approach consists of three specifications describing the fault-tolerant system, the errors to be detected, and the actions to take to recover from those errors. System specification utilizes an object-oriented database to store the descriptions associated with these large, complex systems, while the error detection and error recovery specifications are defined using the formal specification notation Z. We also describe a novel implementation architecture and explore our solution through the use of two case study applications.

## **Keywords**

Critical infrastructures, distributed information systems, survivability, fault tolerance.

# FAULT TOLERANT DISTRIBUTED INFORMATION SYSTEMS

## 1. Introduction

We present an approach to the specification and implementation of fault tolerance in distributed information systems. The approach is based on the use of an enhanced system architecture together with software that is synthesized from a formal specification. The resulting system is designed to deal with a wide variety of fault types and thereby provide a means to improve the dependability of these systems.

By a distributed information system, we mean one that operates on a distributed target and that requires the correct operation of more than one node to provide useful service. Thus, a set of interconnected routers that implement an e-mail system for a set of clients is a distributed information system by our definition, but a set of interconnected workstations that merely share peripherals is not.

Many distributed information systems operate on very large networks, often with many thousands of nodes. Frequently the services these systems provide are critical to their users, and this criticality leads to requirements for very high levels of system dependability [18]. To improve dependability, designers can employ traditional techniques such as N-modular redundancy at the component level in processing, communications, data storage, power supplies, and so on. These techniques cope well with a wide variety of faults, and they can be extended to include replication of complete network nodes if appropriate. However, the expense of complete node replication is often prohibitive.

In this research, we are not concerned with faults that affect a single hardware or software component or even a complete network node. We refer to such faults as *local*, and, although they are important, we assume that all local faults are dealt with by redundant components that mask their effects. For systems for which dependability is important and for which local faults are expected to occur at an unacceptable rate, the effects of local faults are usually masked.

We are concerned with the need to tolerate faults that affect significant fractions of a network, faults that we refer to as *non-local*. Coordinated security attacks are examples of non-local faults. They generally involve attacks on several nodes in a network in a coordinated manner so as either to cause more damage or to gain more illegal access than would be possible with an attack on a single node. Determination that such an attack is underway (using information such as intrusion detection alarms or other anomalies from individual nodes) requires integrated analysis of the state of many elements of the network over perhaps a protracted period of time. Once detected, the treatment of such attacks involves immediate software invocation of additional protection mechanisms such as closing connections and/or files over a wide area together with longer term measures such as changing cryptography keys, user passwords, and so on.

Non-local faults are much more difficult to deal with than local faults. Masking non-local faults requires an unacceptable level of redundancy; the complete replicas of large numbers of nodes could be included in the design of a modern network but the cost would be prohibitive. Our approach to dealing with non-local faults is to use a system architecture, referred to as a *survivability architecture* or an *information survivability control system* [27], in which the network state is monitored, the state is analyzed so that errors can be detected, and network changes are effected to recover from the errors for a prescribed set of faults. These network changes will usually involve changes to the supplied services including reducing some, ceasing others, and perhaps initiating services that are not normally available (such as basic emergency services). Monitoring and change are carried out by sensing and actuating software that resides on network elements of interest. Analysis is performed by servers that are not part of the application network, and communication between the monitored nodes and the servers is by independent communications channels. Survivability architectures are discussed elsewhere [27] and illustrated in Figure 1.

In this paper, we present an approach to the specification and implementation of survivability architectures using a formal specification of the errors of interest and the associated error recov-



Figure 1. Survivability Architecture

ery procedures that are required. We also present an approach to implementation in which the software that implements the survivability architecture is synthesized. This approach is taken because, in the networks of interest, conventional implementation approaches would be too slow and far too prone to error. In section 2, we discuss the characteristics of distributed information systems and the unique characteristics of their faults. In section 3, the goals and our solution approach are presented, and in section 4 the results of preliminary assessment are summarized. Related work is reviewed in section 5 and in section 6, we present our conclusions.

## 2. Critical Distributed Information Systems

The distributed information systems that motivate our research arise in the context of the nation's critical infrastructures. Transportation, telecommunications, power distribution and financial services are examples, and such services have become vital to the normal activities of society. Similarly, systems such as the Global Command and Control System (GCCS) are vital to the nation's defense operations.

In most cases, all or most of the service provided by an infrastructure application can be lost quickly if certain faults arise in its information system. Since societal dependence on critical

infrastructures is considerable, substantial concern about the dependability of the underlying information systems has been expressed, particularly their security [22, 24].

Critical information systems are typically networks with very large numbers of heterogeneous nodes that are distributed over wide geographic areas [16]. It is usually the case that these systems employ commodity hardware, and that they are based on COTS and legacy software. The networks are often private, and implement point-to-point connectivity (unlike the full connectivity of the Internet) because that is all that is required by the associated applications. Within this structure, there are different numbers of the different types of node, and the different types of node provide different functionality. Some nodes provide functionality that is far more critical than others leading to a situation where the vulnerabilities of the information system tend to be associated with the more critical nodes.

An important architectural characteristic of many critical information systems is that provision of service to the end user frequently involves several nodes operating in sequence with each supplying only part of the overall functionality. This *composition of function* can be seen easily in the processing of checks in the nation's financial payment system where the transfer of funds from one account to another involves user-specific actions at local banks, transaction routing by regional banks, bulk funds transfer by central banks, and complex signaling and coordination that assures fiscal integrity[28].

A significant complication that has to be kept in mind is that critical infrastructures are *interdependent*. The power system and the telecommunications network, for example, are critical infrastructures, and they are connected directly to almost all the other critical infrastructures and to each other. The result of this interdependence is that faults in one critical infrastructure can have serious and immediate impact in another.

In critical information systems, there are a number of non-local faults with which we have to be concerned—extensive loss of hardware, failure of an application program, power failures,

security attacks, operator errors, and so forth. These *relatively* simple faults are only a small part of the problem. In practice, fault tolerance is made more complex by the following issues:

- *Fault Sequences*

It is necessary to deal with fault *sequences*, i.e., new faults arising before repairs from earlier faults have been completed. This implies that responses to faults will have to be determined by the overall network state at the time that the fault arises.

- *Fault Hierarchies*

Each non-local fault that arises must be dealt with appropriately. However, once the effects of a fault are detected, the situation might deteriorate leading to the subsequent diagnosis of a more serious fault requiring more extensive error recovery. This leads to the notion of a fault hierarchy. In general, a hierarchy of faults has to be considered both in terms of detection and recovery.

- *Interdependent Network Faults*

Given the interdependent nature of critical networks, the effects of faults in one network that impact a second network must be taken into account in developing approaches to tolerating faults.

### **3. Fault Tolerant System Solution Approach**

Tolerating a fault requires that the effects of the fault be detected, i.e., *error detection*, and that the effects of the fault be dealt with, i.e., *error recovery* [18]. Both of these aspects of fault tolerance are provided, in part, by a monitor/analyze/respond architecture. The specific approaches that are used depend on the particular faults of interest. In this section, we present the issues in error detection and error recovery, and our approach to dealing with them (known as RAPTOR).

The size of critical information systems, the variety and sophistication of the services they provide, and the complexity of the reconfiguration requirements mean that an approach to fault

tolerance that depends upon traditional software development techniques is infeasible in all but the simplest cases. The likelihood is that future systems will involve at least tens of thousands of nodes, have to tolerate dozens, perhaps hundreds, of different types of fault, and have to support applications that provide very elaborate user services. Developing software for a monitor/analyze/respond architecture to implement fault tolerance for such a system using conventional, human-intensive methods is quite impractical, and so the basis of our implementation is direct software *synthesis* from *formal specifications*. This approach also provides an opportunity for analysis of the specification and it facilitates rapid change of the fault-tolerance mechanism in the event that new faults have to be detected or different error recovery approaches are required. This is very likely with security attacks, i.e., deliberate faults.

The overall approach to the implementation of a survivability architecture is shown in Figure 2. The fault-tolerance specification shown on the left of the figure is a set of items, each in a formal language, that describes the network, the requisite error detection and recovery. The translator shown is actually a combination of items that check the specification and synthesize the various elements of the implementation. The structure shown on the right of the figure is the resulting system architecture in which the application network is supplemented with sensing and actuating software, and a separate facility performs analysis.

The fault-tolerance specification is in three parts: System Specification, Error Detection Specification, and Error Recovery Specification. The System Specification defines the system at multiple abstraction levels, grouping nodes according to common characteristics and enabling reference and manipulation of large numbers of nodes at the same time. The Error Detection Specification describes error states for nodes and collection of nodes. Finally, the Error Recovery Specification defines responses to each fault in terms of high-level abstractions of actions provided by the application. In this section, we examine each of these in turn.

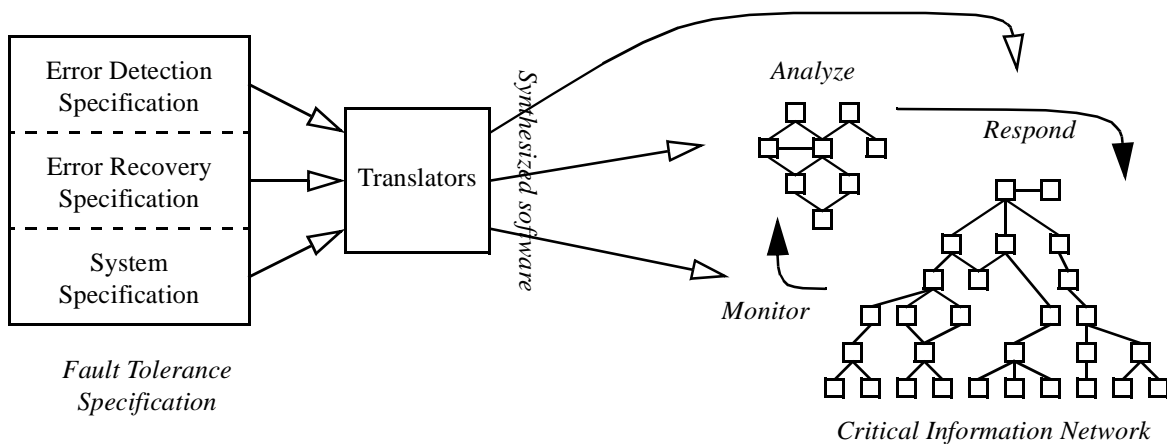


Figure 2. Overall approach

### 3.1 System Specification

In order to specify the expected effects of a fault (and thereby detect it) and the reconfiguration activities needed in response to a fault (and thereby recover from it), complete details of the configuration of the system must be known, and the parts of the system to be reconfigured must be outlined. In essence what is needed is access to the state of the network.

In the RAPTOR system, we use an object-oriented database[23] to hold this essential network state information. C++ is used as the database definition language, and the particular database system used in our implementation provides a preprocessor and compiler to generate its own database schemas and database from the C++ class definition. The C++ class hierarchy used for system specification enables multiple levels of abstraction to be described easily using containment. This hierarchy is precisely what is required to permit the description of compound system objects such as a critical server combination composed of multiple related elements.

Multiple abstraction hierarchies can be specified thereby allowing system objects such as individual nodes to belong to different compound objects based on general characteristics. For example, in a financial system, one abstraction hierarchy might correspond to bank ownership and administrative control, where branch banks are owned by their commercial money-center

banks, and then all the money-center banks are owned by the Federal Reserve Bank. Another abstraction hierarchy, however, would represent the distribution of electric power, where power companies contain references to all the banks that they serve.

### **3.2 Error Detection**

Describing the network state resulting from a non-local fault requires various abstraction mechanisms because of the complexity that arises with many different network elements (nodes, links, software components, etc.). For example, it might be necessary to refer to a set of nodes that have common characteristics such as experiencing the same execution-time event (power failure, software failure, security alarm, etc.), being within the same geographic region, having similar software configurations, and so on. Such sets usually do not need to distinguish specific nodes, merely nodes of a specific type with specific characteristics. Sets of individual nodes might be grouped according to various characteristics, and these sets of nodes manipulated to describe states of interest. Similarly, data on sets of individual nodes can be referenced at higher levels of abstraction, such as a system or global level. We use a set-based notation for state description within a specification because of these considerations.

The actual detection of errors is carried out by a *state machine* since an error (more precisely an erroneous state) corresponds to a network system state of interest. As the effects of a fault manifest themselves, the state changes. The changes become input to the state machine in the form of events, and the state machine signals an error if it enters a state designated as erroneous. The various states of interest are described using predicates on the sets mentioned above. The general form of a specification of an erroneous state, therefore, is a collection of set definitions that identify the network objects of concern and predicates using those sets to describe the various states for which either action needs to be taken or which could lead to states for which action needs to be taken.

In an operating network, events occurring at the level of individual nodes are recognized by a

finite-state machine at what amounts to the lowest level of the system. This is adequate, for example, for a fault like a wide-area power failure. Dealing with such a fault might require no action if the number of affected nodes is below some threshold. Above that threshold might require certain critical network nodes to respond by limiting their activities. As node power failures are reported so a predefined set, *nodes\_without\_power*, is modified, and once its cardinality passes the threshold, the recognizer moves to an error state.

The notion of a fault hierarchy requires that more complex fault circumstances be recognized. A set of nodes losing power in the West is one fault, a set losing power in the East is a second, but both occurring in close temporal proximity might have to be defined as a separate, third fault of much more significance because it might indicate a coordinated terrorist attack. This idea is dealt with by a *hierarchy* of finite-state machines. Compound events can be passed up (and down) the hierarchy, so that a collection of local events can be recognized at the regional level as a regional event, regional events could be passed up further to recognize national events, and so on. As an example, a widespread coordinated security attack might be defined to be an attack within some short period of time on several collections of nodes each within a separate administrative domain. Detection of such a situation requires that individual nodes recognize the circumstances of an attack, groups of nodes collect events from multiple low-level nodes to recognize a wide-area problem, and the variety of wide-area problems along with their simultaneity recognized as a coordinated attack.

Figure 3 shows this notion of hierarchical finite state machines together with our treatment of interdependent networks. Events from one network survivability mechanism can be transmitted to other survivability mechanisms in the form of compound events thereby permitting a proactive response in a network that has not itself sustained any damage.

RAPTOR uses a subset of the formal specification notation  $Z$  to describe the finite-state machines for error detection. As a state-based specification language,  $Z$  enables finite-state

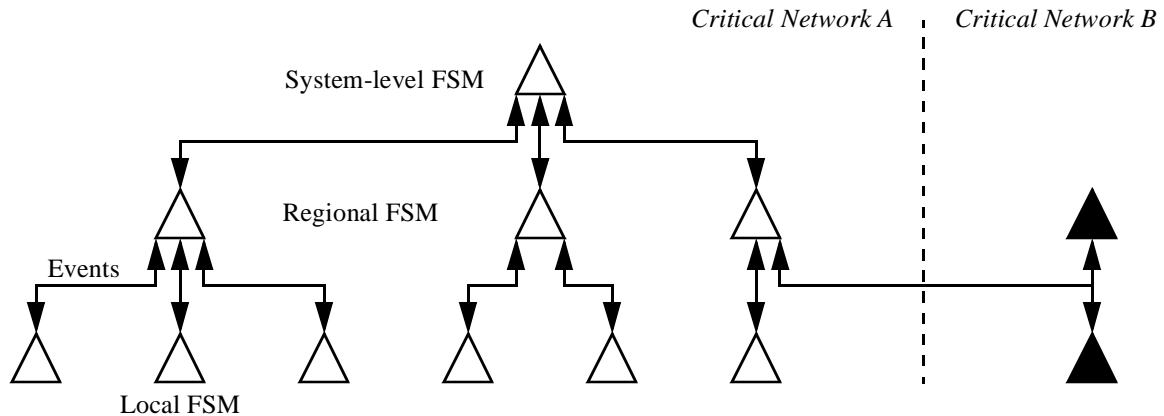


Figure 3. Hierarchy of finite-state machines.

machines to be defined formally and in a very straightforward manner. RAPTOR includes a translator that synthesizes a C++ implementation from a specification written in the Z subset. Generation of executable code for finite-state machines is also straightforward. A Z state description schema is defined for each finite-state machine, including any invariants and state variables from the system specification. Events for finite-state machines at each level of the abstraction hierarchy (e.g., node, region, system) are declared as messages, and schemas for each event are defined to specify that event's effect on the state.

### 3.3 Error Recovery

Error recovery for a fault whose effects cannot be masked requires that the network application itself be reconfigured. The goal of reconfiguration is to effect changes such as terminating, modifying, or moving certain running application elements, and starting new applications. In a banking application, for example, it might be necessary to terminate low priority services such as on-line customer enquiry, modify some services such as limiting electronic funds transfers to corporate customers, and start emergency services such as providing crucial operational data to the Federal Reserve Bank.

Unless provision for reconfiguration is made in the design of the application, reconfiguration

will be a dhoc at best and impossible at worst [15]. The provision for reconfiguration in the application design has to be quite extensive in practice for three reasons:

- The number of fault types is likely to be large and each might require different actions following error detection.
- It might be necessary to complete reconfiguration in bounded time so as to ensure that the replacement service is available in a timely manner.
- Reconfiguration itself must not introduce new security vulnerabilities.

Just what is required to permit application reconfiguration depends, in large measure, on the design of the application itself. Provision must be made in the application design to permit the service termination, initiation, and modification that is required by the specified fault-tolerant behavior. The RAPTOR Error Recovery Specification defines at a high level the activities for each fault in the Error Detection Specification. Because faults correspond to transitions in the finite-state machines, error recovery actions are defined for each transition into an erroneous state.

The application must be constructed so as to provide for the error recovery actions required in response to the prescribed faults. The application architecture that RAPTOR assumes is a collection of interacting node programs each of which consists of a set of cooperating sequential processes. The individual processes must include provision for a set of crucial actions that might be required during error recovery together with an interface that permits the actions to be invoked. This set of mandatory crucial actions includes, for example, “stop” which, if it is invoked, requires a process to reach a quiescent state in bounded time.

The Error Recovery Specification refers to these various actions as a set of messages that can be sent to application nodes. To specify error recovery, the Error Recovery Specification defines high-level sequences of actions that the application is to undertake for each set of recovery activities in response to faults.

The actions provided by the application are declared in the System Specification for each node and set of nodes in the abstraction hierarchy. The errors, of course, are defined in the Error Detection Specification and correspond to transitions in the finite-state machines. Thus, the three specification notations are integrated and fully specify the necessary aspects of fault tolerance.

Application-related action sequences are defined for appropriate state transitions, but the actions have to be tailored to both the final (erroneous) state of the transition and the *initial* state. This is the way in which sequential faults, i.e., a bad situation that gets worse, are handled. For example, a wide-area power failure has to be handled very differently if it occurs in a benign state than if it occurs following a previous traumatic loss of computing equipment perhaps associated with a terrorist attack.

#### **4. Preliminary Evaluation**

If the approach we have described is of any value, then it has to tolerate non-local faults in systems with the characteristics discussed in section 2. In order to evaluate our approach, at least in part, we undertook two case studies. One involved a hypothetical financial payments system and the second involved a version of the nation's electric power grid.

For the purposes of exploring the specifications and testing the synthesized implementations, we used simulation models since, in general, it is impossible to test on typical critical information systems. The RAPTOR modeling system provides facilities to: (1) simulate tens of thousands of nodes of any number of types and with arbitrary functionality for each node type; (2) describe and effect arbitrary network architectures and topologies; (3) associate prescribed vulnerabilities with any nodes in the model; and (4) inject failures at nodes according to parameterized scripts.

##### **4.1 Scenario Descriptions**

The first case study that we used was a hypothetical financial payments network. The functionality of this example is a vastly simplified version of the United States financial payments system,

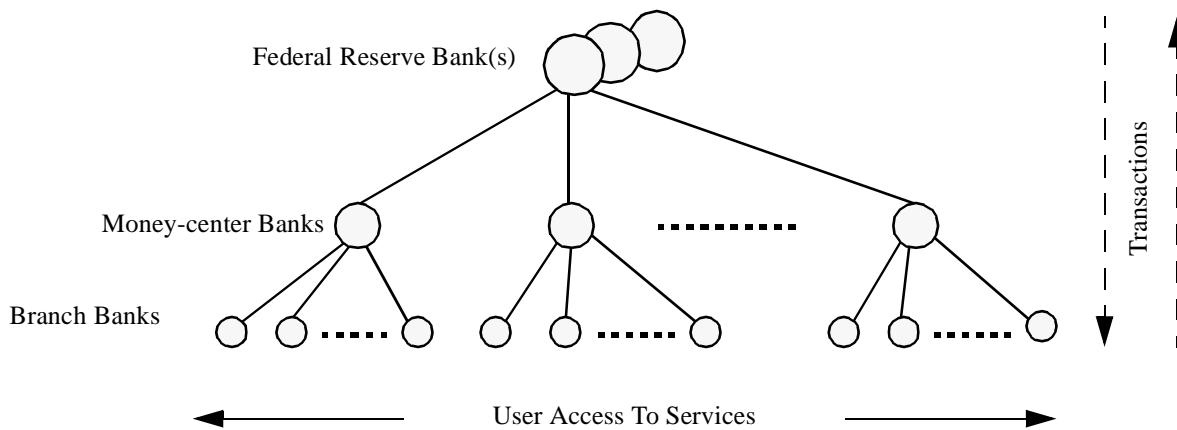


Figure 4. Hypothetical payments network.

where value transfer is effected by the routing of funds throughout the banking network. The system architecture (shown in Figure 4) consists of a three-level hierarchy. The model contains a Federal Reserve Bank node with two synchronized (hot spare) backup nodes, 100 commercial money-center banks, and 9900 branch banks with roughly 100 branch banks for each money-center bank. The branch banks provide customer service access to all financial services. Money-center banks are the service centers for commercial operations, and they host customer accounts, route payment requests between their branches and batch payment requests destined for other commercial banks. The most important nodes in the system by a considerable degree belong to the Federal Reserve Bank since it actually transfers funds between retail commercial banks. A distributed control system was added that has appropriate connections to the application network allowing the necessary monitoring, analysis, and response.

We defined a set of local faults at each node and level of the abstraction hierarchy. For each bank node, local faults included: intrusion detection alarm triggered, database failure, power failure, and total site failure. More importantly, our evaluation was based on a set of non-local faults that were defined using combinations of local faults. The non-local faults that were defined were:

- coordinated security attacks in which intrusion detection alarms were triggered on different

combinations of nodes within certain time periods,

- cascading software failures in which a software failure at one node leads to failure at connected nodes and their connected nodes and so on, and
- regional power failures.

A set of application responses were defined for each of the prescribed faults. In practice, a systems engineer in the application domain would determine appropriate responses to each fault of concern; for purposes of our experimentation we defined what we considered reasonable responses to prove the feasibility of some response.

The electric power system model was based on the power grid information systems, i.e., the information systems associated with power generation and distribution in the United States electric power grid [29]. The model consists of the Eastern, Western, and the ERCOT Interconnections, ten control regions, 143 control areas, 429 power companies, 1,287 generating stations, and 2,574 substations. In the model, each control area is responsible for three power companies, and each power company manages three generating facilities and six major substations. The substations in the model signify demand for power, while the generators represent power supply.

The low-level faults designed for the power system were intrusion detection alarm, database failure, and full node failure. Complex, non-local faults for this model included coordinated security attacks, the loss of significant generating power in a power company or control area, and the failure of key control area or control region nodes. Responses were defined for each fault of concern, including increased generation to compensate for power loss, local balancing functions, and backup processing capacity.

Given this application model, a control system to monitor these information system nodes was constructed. The hierarchical control system corresponds roughly to the administrative and regulatory hierarchies in place in the electric power grid. In fact, two overlapping control system hier-

archies were modelled: one for general information system monitoring and the other for intrusion and security monitoring. The first, corresponding to the administrative hierarchy, consisted of finite-state machines at each power company, control area, and control region. The second, corresponding to the NERC hierarchy for system security issues, consisted of finite-state machines at each control area, control region, and the National Infrastructure Protection Center (NIPC).

#### **4.2 Banking System Specification and Implementation**

For the banking system model, the System Specification defined each bank node type using a class in the database schema. The Error Detection Specification contained three Z state schemas to model the three levels of the banking hierarchy together with a set of operation schemas to specify the transitions of the finite-state machines on occurrence of faults. For the branch bank (local level) state schema, there were 19 operation schemas; for the money-center bank (regional level) state schema, there were 27 operation schemas; and for the Federal Reserve (system level) state schema, there were 23 operation schemas. The Error Recovery Specification consisted of a set of message definitions: the branch bank required 8 messages to specify alternate service modes; the money-center bank required 10 messages, and the Federal Reserve bank required 10 messages.

The Z specifications were translated into C++ by the RAPTOR translator. This software created the analysis component of the system, and the monitoring and response supplements that were needed for the individual nodes. This software was integrated into the C++ code synthesized for the database system to produce a complete implementation that was executed using the modeling system.

#### **4.3 Results of the Case Studies**

The results of three example experiments in which non-local faults were injected into our two models are shown in Figures 5, 6, and 7. The first two are for the financial system model and the third is for the power grid model. The data shown is for illustration and is for a single simulation.

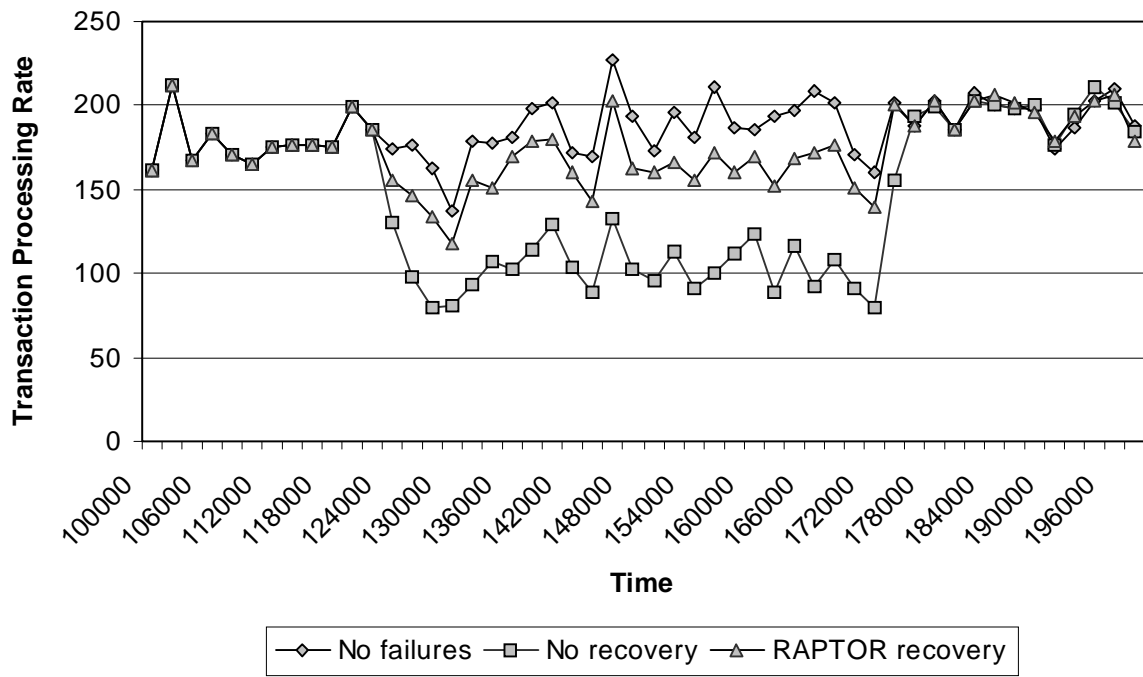


Figure 5. Loss of 25% of the money-center banks.

Results of more extensive experimentation including average data is available elsewhere re[9].

In Figure 5, the injected fault was a sudden loss of 25% of the mid-level money-center banks as might occur with a common-mode software failure or the corruption of databases because of common erroneous transactions. The recovery specified in this case is to require that a designated branch bank take over as a backup. That bank then has to terminate all local services and start software to implement money-center bank service at a reduced level. With no recovery total system transaction rates are vastly reduced. With reconfiguration, the impact is considerably less.

The fault in Figure 6 is corruption of the databases at the primary Federal Reserve processing center. The recovery that is effected is for each commercial bank to merely queue all its transactions during the outage. These transactions are then processed when the fault is corrected. In the figure it can be seen that processing drops to almost zero during the fault and rises above the steady state after the fault is corrected to accommodate the queued transactions.

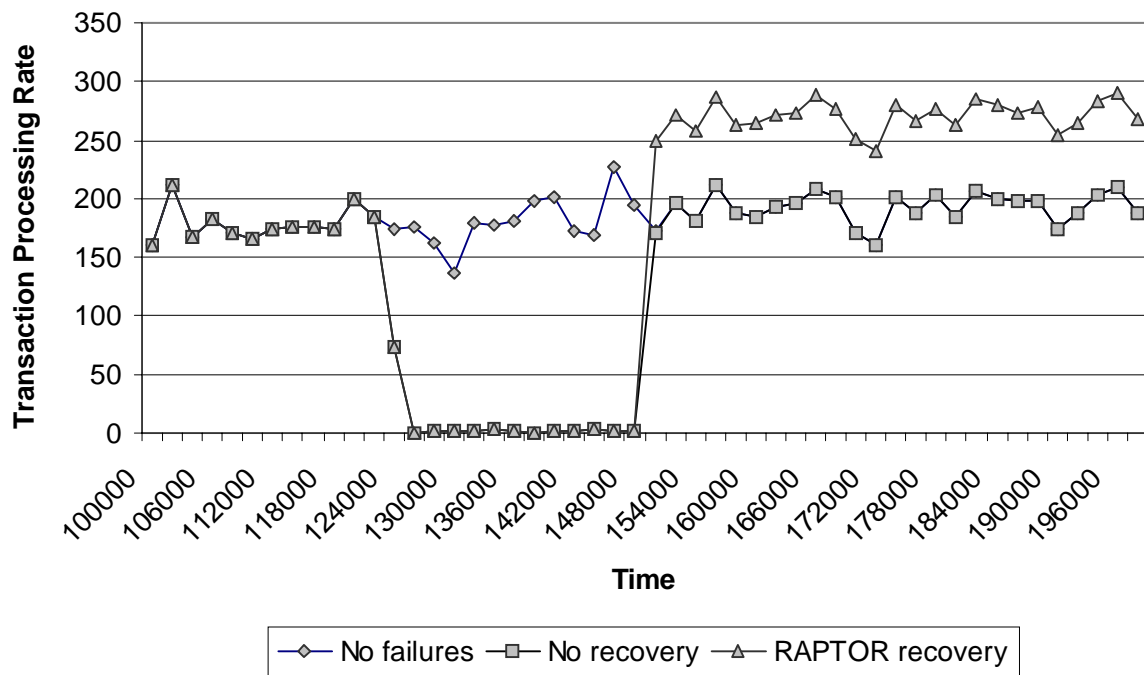


Figure 6. Federal Reserve Bank database failure)

The results of the third experiment are shown in Figure 7. In this case, 5% of the system’s generating capacity is lost and this followed sometime later by the loss of a further 5%. The recovery mechanism that was employed in this experiment was to raise the power generated by remaining equipment within each operating region to the highest possible level and supply what is available just to the region producing it.

The primary result that we have obtained from these case studies is to demonstrate feasibility of the specification approach, the architecture, and software synthesis. For two domains, we have modeled realistic information systems each with very large numbers of nodes, representative topologies, and feasible error recovery requirements. Error detection and recovery requirements were defined completely using the RAPTOR notations (including Z) and all of the necessary operational software synthesized. Significant non-local faults were injected into the models and the resulting systems shown to perform much more effectively than the unmodified versions.

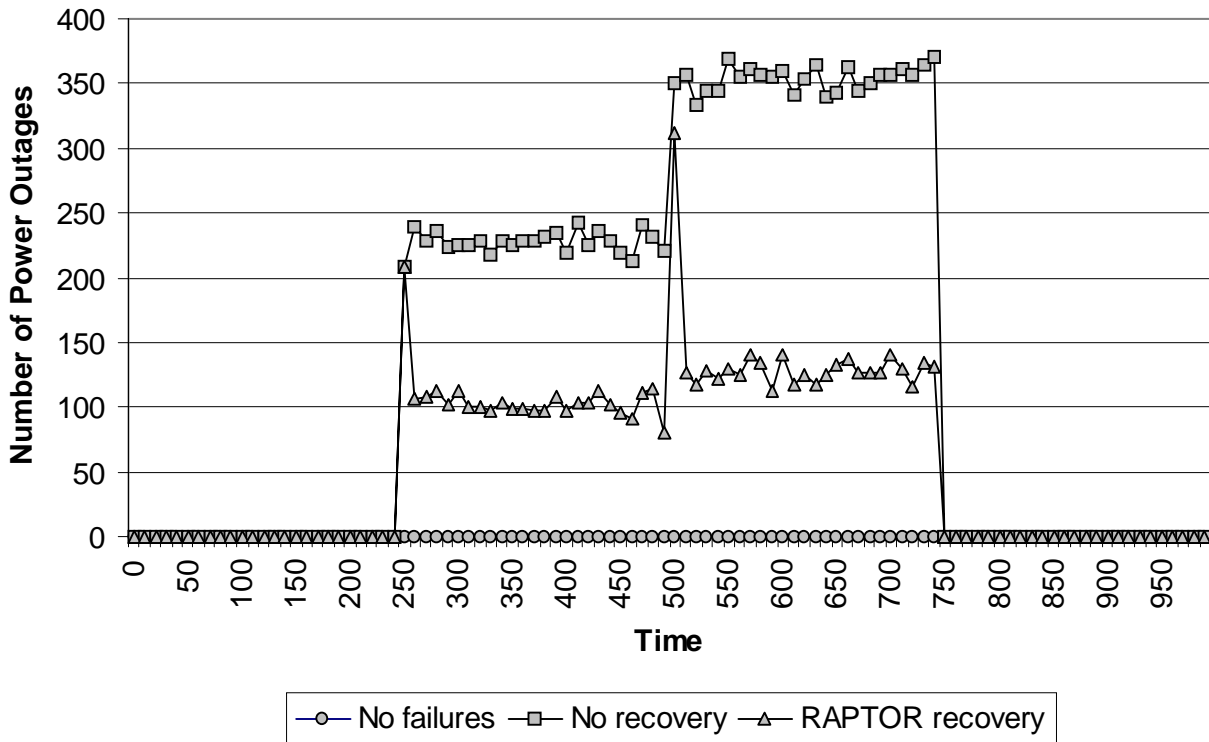


Figure 7. Five percent then ten percent generator failures.

## 5. Related Work

System-level approaches to fault tolerance are described by Birman [5] and Jalote [14]. Cristian provides a survey of the issues involved in providing fault-tolerant distributed systems [7], and an instantiation of Cristian's fault tolerance concepts was used in the Advanced Automation System (AAS) [8]. Similar work by Birman et al. on group communication systems manages the redundancy in groups of processes in order to mask processor failures [4]. While these efforts and others [3, 26] are significant in their own right, they do not address non-local faults, and they tend not to address systems as large and complex as critical information systems.

Marzullo and Alvisi are concerned with replication strategies for large-scale distributed applications with dynamic (unpredictable) communication properties and a requirement to withstand security attacks [1]. Other work by Melliar-Smith and Moser attempts to provide transparent fault

tolerance to users in the Eternal system, middle ware that operates in a CORBA environment, below a CORBA ORB but on top of their Totem group communication system[21].

CONIC, a language and distributed support system, was developed to support dynamic configuration [17]. Darwin is a configuration language that separates program structure from algorithmic behavior [20]. Darwin utilizes a component- or object-based approach to system structure in which components encapsulate behavior behind a well-defined interface. Darwin is a declarative binding language that enables distributed programs to be constructed from hierarchically-structured specifications of component instances and their interconnections [19].

Purtilo and Hofmeister studied the types of reconfiguration possible within applications and the requirements for supporting reconfiguration [13]. They integrated additional reconfiguration primitives into the Polyolith Software Bus [25], leveraging off of Polyolith's interfacing and message-passing facilities in order to ensure state consistency during reconfiguration[12].

## **6. Conclusions**

Fault tolerance in critical information systems is essential because the services that such systems provide are crucial. In attempting to deal with faults in such systems, it becomes clear immediately that the complexity of the fault-tolerance mechanism itself could be a serious liability for the system. The number of system states and the number of possible faults are such that the creation of a fault-tolerant system using typical hand-crafted development is infeasible.

We have developed a specification-based approach that deals with the problem by reducing it to the creation of a formal specification from which an implementation is synthesized. The complexity of the specification itself is reduced significantly by using a variety of abstractions. We note that the implementation issues which arise in the approach that we have described are very significant but are not addressed in this paper.

The overall approach permits fault tolerance to be introduced into networks in a manageable way. The detailed utility of the approach is presently under investigation as are the mechanics of

implementation.

## **Acknowledgements**

This effort sponsored in part by the Defense Advanced Research Projects Agency under agreement F30602-99-1-0538 and in part by the Defense Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-96-1-0314. The U.S. Government is authorized to reproduce and distribute reprints for governmental purpose notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory or the U.S. Government.

## **References**

- [1] Alvisi, L. and K. Marzullo. "WAF: Support for Fault-Tolerance in Wide-Area Object Oriented Systems," Proceedings of the 2nd Information Survivability Workshop, IEEE Computer Society Press, Los Alamitos, CA, October 1998, pp. 5-10.
- [2] Anderson, T. and P. Lee. Fault Tolerance: Principles and Practice. Prentice Hall, Englewood Cliffs, NJ, 1981.
- [3] Bhatti, N., M. Hiltunen, R. Schlichting, and W. Chiu. "Coyote: A System for Constructing Fine-Grain Configurable Communication Services," ACM Transactions on Computer Systems, Vol. 16 No. 4, November 1998, pp. 321-366.
- [4] Birman, K. "The Process Group Approach to Reliable Distributed Computing," Communications of the ACM, Vol. 36 No. 12, December 1993, pp. 37-53 and 103.
- [5] Birman, K. Building Secure and Reliable Network Applications. Manning, Greenwich, CT, 1996.
- [6] Cowan, C., L. Delcambre, A. Le Meur, L. Liu, D. Maier, D. McNamee, M. Miller, C. Pu, P. Wagle, and J. Walpole. "Adaptation Space: Surviving Non-Maskable Failures," Technical Report 98-013, Department of Computer Science and Engineering, Oregon Graduate Institute of Science and Technology, May 1998.
- [7] Cristian, F. "Understanding Fault-Tolerant Distributed Systems," Communications of the ACM, Vol. 34 No. 2, February 1991, pp.56-78.

- [8] Cristian, F., B. Dancey, and J. Dehn. "Fault-Tolerance in Air Traffic Control Systems," ACM Transactions on Computer Systems, Vol. 14 No. 3, August 1996, pp. 265-286.
- [9] Elder, M.C. "Fault Tolerance in Critical Information Systems," Ph.D. dissertation, Department of Computer Science, University of Virginia, May 2001.
- [10] Ellison, B., D. Fisher, R. Linger, H. Lipson, T. Longstaff, and N. Mead. "Survivable Network Systems: An Emerging Discipline," Technical Report CMU/SEI-97-TR-013, Software Engineering Institute, Carnegie Mellon University, November 1997.
- [11] Gartner, Felix C. "Fundamentals of Fault-Tolerant Distributed Computing in Asynchronous Environments," ACM Computing Surveys, Vol. 31 No. 1, March 1999, pp.1-26.
- [12] Hofmeister, C., E. White, and J. Purtilo. "Surgeon: A Packager for Dynamically Reconfigurable Distributed Applications," Software Engineering Journal, Vol. 8 No. 2, March 1993, pp. 95-101.
- [13] Hofmeister, C. "Dynamic Reconfiguration of Distributed Applications," Ph.D. Dissertation, Technical Report CS-TR-3210, Department of Computer Science, University of Maryland, January 1994.
- [14] Jalote, P. Fault Tolerance in Distributed Systems. Prentice Hall, Englewood Cliffs, NJ, 1994.
- [15] Knight, J., M. Elder, and X. Du. "Error Recovery in Critical Infrastructure Systems," Proceedings of Computer Security, Dependability, and Assurance 1998, IEEE Computer Society Press, Los Alamitos, CA, 1999, pp. 49-71.
- [16] Knight, J., M. Elder, J. Flinn, and P. Marx. "Summaries of Three Critical Infrastructure Systems," Technical Report CS-97-27, Department of Computer Science, University of Virginia, November 1997.
- [17] Kramer, J. and J. Magee. "Dynamic Configuration for Distributed Systems," IEEE Transactions on Software Engineering, Vol. SE-11 No. 4, April 1985, pp. 424-436.
- [18] Laprie, Jean-Claude. "Dependable Computing and Fault Tolerance: Concepts and Terminology," Digest of Papers FTCS-15: 15th International Symposium on Fault-Tolerant Computing, pp. 2-11, 1985.
- [19] Magee, J., N. Dulay, S. Eisenbach, and J. Kramer. "Specifying Distributed Software Architectures," Lecture Notes in Computer Science, Vol. 989, September 1995, pp. 137-153.
- [20] Magee, J. and J. Kramer. "Darwin: An Architectural Description Language," <http://www.dse.doc.ic.ac.uk/research/darwin/darwin.html>, 1998.
- [21] Melliar-Smith, P. and L. Moser. "Surviving Network Partitioning," IEEE Computer, Vol. 31 No. 3, March 1998, pp.62-68.
- [22] Office of the Undersecretary of Defense for Acquisition and Technology. "Report of the Defense Science Board Task Force on Information Warfare - Defense (IW-D)," November 1996.

[23] Poet Software Corp. <http://www.poet.com>

[24] President's Commission on Critical Infrastructure Protection. "Critical Foundations: Protecting America's Infrastructures The Report of the President's Commission on Critical Infrastructure Protection," United States Government Printing Office (GPO), No. 040-000-00699-1, October 1997.

[25] Purtilo, J. "The POLYLITH Software Bus," ACM Transactions on Programming Languages and Systems, Vol. 16 No. 1, January 1994, pp.151-174.

[26] Shrivastava, S., G. Dixon, G. Parrington. "An Overview of the Arjuna Distributed Programming System," IEEE Software, Vol. 8 No. 1, January 1991, pp.66-73.

[27] Sullivan, K., J. Knight, X. Du, and S. Geist. "Information Survivability Control Systems," Proceedings of the 21st International Conference on Software Engineering, IEEE Computer Society Press, Los Alamitos, CA, May 1999, pp.184-192.

[28] Summers, B. The Payment System: Design, Management, and Supervision. International Monetary Fund, Washington, DC, 1994.

[29] NERC operating manual from January 2001