APPLICATION ERROR RECOVERY IN CRITICAL INFORMATION SYSTEMS

John C. Knight **Matthew C. Elder**

Department of Computer Science
University of Virginia

151 Engineer's Way, P.O. Box 400740
Charlottesville, VA 22904-4740

August 25, 2000

knight@cs.virginia.edu *elder@cs.virginia.edu*

(804) 982-2216

Abstract

Critical infrastructure applications 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. Fault tolerance consists of two primary stages, error detection and error recovery; in this paper we focus on application error recovery in these critical information systems. We outline a specification-based approach to error recovery that enables systematic structuring of error recovery specifications, an implementation partially synthesized from the formal specification, and various forms of static and run-time analysis. We present the RAPTOR specification notation for describing error recovery activities in the face of various faults, and we explore synthesis of implementation code using the Error Recovery Translator. We also describe a novel implementation architecture enabling error recovery in these systems and discuss issues in analysis.

APPLICATION ERROR RECOVERY IN CRITICAL INFORMATION SYSTEMS

1 Introduction

Our dependence on large infrastructure systems has increased to a point where the loss of the services that they provide is extremely disruptive [26], [27]. Infrastructures such as transportation, telecommunications, power distribution, and financial services are absolutely vital to the normal operation of society. Similarly, systems such as the Global Command and Control System (GCCS) are vital to defense operations. We refer to such applications as *critical infrastructure applications.*

These applications are themselves dependent on complex, distributed information systems, and all or most of the service provided by an infrastructure application can be lost quickly if faults arise in its information system. We refer to such information systems as *critical information systems.*

Having to deal with faults that might disrupt service in information systems leads to the notion of *survivability* [17]. Informally by survivability we mean the ability of the system to continue to provide service (possibly degraded) when serious traumas occur in the system or the operating environment. For example, when faults such as extensive hardware failure, software failure, operator error, or malicious attack occur, a critical subset of normal functionality or some alternative functionality might be needed to mitigate the consequences of the fault.

Survivability is a system *requirement.* The particular survivability requirements that must be met are described in a survivability specification that is a statement of the prioritized list of service levels that the system must provide and the associated probability for each service level being provided. There is no presumption about how survivability will be achieved in the notion of survivability itself. One essential aspect of system design is to ensure that systems are built to satisfy the probabilistic service requirements dictated by the survivability specification.

There are many mechanisms or strategies that can be employed to achieve survivability requirements in the face of faults in the system or changes in the system environment. The process of building a system in such a way that certain faults do not arise is *fault avoidance.* Building systems that are able to react in a requisite way to prescribed faults when they do arise is *fault tolerance* [2], [11]. In practice, the use of fault tolerance is essential in the types of heterogeneous distributed systems that underlie critical infrastructure applications. Fault tolerance breaks down into two general stages: first, detecting the effects of the fault, i.e., *error detection*, and second, dealing with the effects of the fault, i.e., *error recovery* [20].

In this document we summarize the issues involved in achieving error recovery in critical information systems. In Section 2, we describe some of the characteristics of the application programs with which we are concerned, critical information systems. The following section provides a more detailed description of the faults with which we are concerned and some fault tolerance definitions. Section 4 presents a motivating example, while the next section outlines an overview of our solution approach. A key component of our solution approach involves formal specification and synthesis of error recovery in these systems; Section 6 goes into great detail exploring

that aspect of the problem. Section 7 describes possible system architectures supporting error recovery, and after that we explore possible analyses that can be performed given formal specifications. Finally, we conclude with an overview of related work and a series of appendices outlining our work on this topic.

2 Critical Information Systems

The President's Commission on Critical Infrastructure Protection cited a variety of infrastructure application domains that have become exceedingly dependent upon their information systems for correct and efficient operation, including banking and finance, various transportation industries, electric power generation and distribution, and telecommunications [27]. Detailed descriptions of four of these applications can be found elsewhere [16].

The architecture of the information systems upon which critical infrastructure applications rely are tailored substantially to the services of the industries which they serve and influenced inevitably by cost-benefit trade-off's. For example, though these systems are typically distributed over a very wide (geographically dispersed) area with large numbers of nodes, the application dictates the sites and the distribution of nodes at those locations. Beyond this, however, there are a variety of similar characteristics possessed by these critical information systems across all application domains that are pertinent to achieving the requirement of enhanced survivability using new error recovery techniques:

- *Heterogeneous nodes.* Despite the large number of nodes in many of these systems, a small number of nodes are often far more critical to the functionality of the system than the remainder. This occurs because critical parts of the system's functionality are implemented on just one or a small number of nodes. Heterogeneity extends also to the hardware platforms, operating systems, application software, and even authoritative domains.

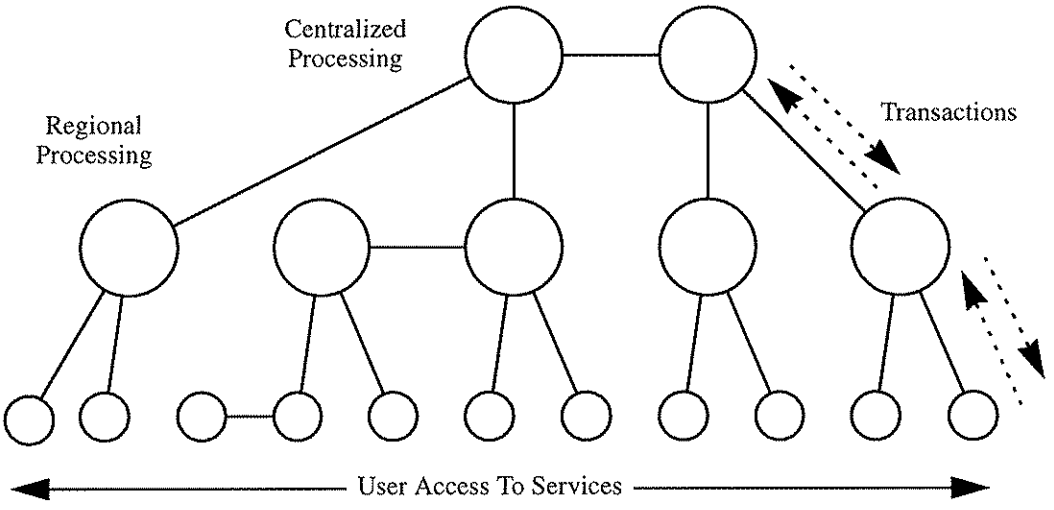

Figure 1. Example application network topology.

- *Composite functionality.* The service supplied to an end user is often attained by composing different functionality at different nodes. Thus, entirely different programs running on different nodes provide different services, and complete service can only be obtained when several subsystems cooperate and operate in some predefined sequence. This is quite unlike more familiar applications such as mail servers routing mail through the Internet. An example network topology for one of these applications is shown in Figure 1.
- *Stylized communication structures.* In a number of circumstances, critical infrastructure applications use dedicated, point-to-point links rather than fully-interconnected networks. Reasons for this approach include meeting application performance requirements, better security, and no requirement for full connectivity.
- *Performance requirements.* Some critical information systems, such as the financial payment system, have soft real-time constraints and throughput requirements (for checks cleared per second, for example), while others, such as parts of many transportation systems and many energy control systems, have hard real-time constraints. In some systems, performance requirements change with time as load or functionality changes—over a period of hours in financial systems or over a period of days or months in transportation systems, for example.
- *Security requirements.* Survivability is concerned with malicious attacks as well as failures caused by hardware and software faults. Given the importance of critical infrastructure applications, their information systems provide an attractive target to terrorists and other parties hostile to the nation intent on disrupting and sabotaging daily life. The deliberate faults exploited by security attacks are of significant concern to an error recovery strategy.
- *Extensive databases.* Infrastructure applications are concerned primarily with data. Many employ several extensive databases with different databases being located at different nodes and with most databases handling very large numbers of transactions.
- *COTS and legacy components.* For all the usual reasons, critical infrastructure applications utilize COTS components including hardware, operating systems, network protocols, database systems, and applications. In addition, these systems contain legacy components—custom-built software that has evolved with the system over many years.

The characteristics listed above are important, and most are likely to remain so in systems of the future. But the rate of introduction of new technology into these systems and the introduction of entirely new types of application is rapid, and these suggest that error recovery techniques must take into account the likely characteristics of future systems as well. We hypothesize that the following will be important architectural aspects of future infrastructure information systems:

- *Larger numbers of nodes.* The number of nodes in infrastructure networks is likely to increase dramatically as enhancements are made in functionality, performance, and user access. The effect of this on error recovery is considerable. In particular, it suggests that error recovery will have to be regional in the sense that different parts of the network will require different recovery strategies. It also suggests that the implementation effort involved in error recovery will be substantial because there are likely to be many regions and there will be many different anticipated faults, each of which might require different treatment.

- *Extensive, low-level redundancy.* As the cost of hardware continues to drop, more redundancy will be built into low-level components of systems. Examples include mirrored disks and redundant server groups. This will simplify error recovery in the case of low-level faults; however, catastrophic errors will still require sophisticated recovery strategies.
- *Packet-switched networks.* For many reasons, the Internet is becoming the network technology of choice in the construction of new systems, in spite of its inherent drawbacks (e.g., poor security and lack of performance guarantees). However, the transition to packet-switched networks, whether it be the current Internet or virtual-private networks implemented over some incarnation of the Internet, seems inevitable and impacts solution approaches for error recovery.

3 Faults and Fault Tolerance

We are concerned in this research with the need to tolerate faults that affect significant fractions of a network application, faults that we refer to as *non-local*. Thus, for example, a widespread power failure in which many application nodes are forced to terminate operation is a non-local fault. The complete failure of a single node upon which many other nodes depend would also have a significant non-local effect and would also be classified as a non-local fault.

Non-local faults have the important characteristic that they are usually *non-maskable*—that is, their effects are so extensive that normal system service cannot be continued with the resources that remain, even if the system includes extensive redundancy [7]. We are not concerned with faults at the level of a single hardware or software component. We refer to such faults as *local*, and we assume that all local faults are dealt with by some mechanism that masks their effects. Thus synchronized, replicated hardware components are assumed so that losses of single processors, storage devices, communications links, and so on are masked by hardware redundancy.

Non-local faults might affect a related subset of nodes in a network application leading to the idea that they can be *regional*. Thus, a fault affecting all the nodes in the banking system in a given city or state would be regional and dealing with such a fault might depend on the specific region that was affected. It is also likely that the elements of a non-local fault would manifest themselves over a period of time rather than instantly. This leads to the notion of *cascading* faults in which application components fail in some sequence over a possibly protracted period of time. Detecting such a situation and diagnosing the situation correctly is a significant challenge.

As mentioned previously, tolerating a fault requires first that the effects of the fault be detected, i.e., *error detection*, and second that the effects of the fault be dealt with, i.e., *error recovery* [20]. The mechanism that we employ to implement fault tolerance is a *survivability architecture* (discussed in detail elsewhere [33]). Both error detection and error recovery have to be defined precisely if a fault-tolerant system is to be built and operated correctly, and several issues arise in dealing with them.

3.1 Error Detection

Error detection for a non-local fault requires the collection of information about the state of the network application and subsequent analysis of the information. Analysis is required to permit a conclusion about the underlying fault to be made given a spectrum of specific information.

The key problem that arises in dealing with error detection in large distributed systems is

defining precisely what circumstances are of interest. Events will occur on a regular basis that are associated with faults that are either masked or of no interest. These events have to be filtered and incorporated accurately in the detection of errors of interest. The possibility of false positives, false negatives, and erroneous diagnosis is considerable. In a banking system, for example, it is likely to be the case that local power failures are masked routinely yet, if a series of local failures occurs in sequence, they are probably part of a widespread cascading failure that needs to be addressed either regionally or nationally.

3.2 Error Recovery

Error recovery for a fault whose effects cannot be masked requires that the application be reconfigured following error detection. The goal of reconfiguration is to effect changes such as terminating, modifying, or moving certain running applications, 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, and modify some services, such as limiting electronic funds transfers to corporate customers.

Unless provision for reconfiguration is made in the design of the application, reconfiguration will be ad hoc 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.

4 Illustrative Example

To illustrate some of the issues that arise in implementing fault tolerance, consider an extremely simple example that is part of a hypothetical financial network application.

The system architecture, shown in Figure 2, consists of a 3-node network with one money-center bank (N1), two branch banks (N2 and N3), and three databases (DB), one attached to each node. There are two full-bandwidth communications links (L1 and L2) and two low-bandwidth backup links (l1 and l2). There is a low-bandwidth backup link between the two branch banks (l3). The intended functionality of this system is implement a small financial payments system, effecting value transfer between customer accounts. In a system free of faults, the branch banks provide customer access (check deposit facilities) and local information storage (customer accounts), while the money-center bank maintains branch bank asset management and switching facilities for check clearance.

The faults with which we would be concerned in a system of this type would be the loss of a

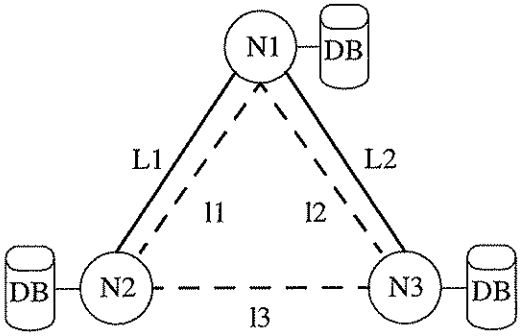

Figure 2. Example fault-tolerant distributed system.

computing node's hardware, the loss of an application program on a node, the loss of a database, the loss of a communications link, and so on. For each of these, it would be necessary first to define the fault and then, for each fault, document what the system is to do if the fault arises. In this example, losing the money-center bank would severely limit customer service since a branch bank would have to take over major services using link l3 for communication. Loss of either of the full-bandwidth communications links would also drastically cut service since communication would have to use a low-bandwidth link.

To implement a fault-tolerant system, the application must be constructed with facilities to tolerate the effects of the particular faults of concern. In the system architecture, the three low-bandwidth communications links provide alternate service in case of failures in the full-bandwidth communication links (L1 and L2) or the primary routing node (N1). The applications themselves must also provide alternate service in case of certain faults; for example, while the primary functionality of the money-center bank N1 is to route deposited checks for clearance and maintain the balances of each branch bank, additional services that can be provided include buffering of check requests for a failed branch bank or acceptance of checks for deposit if the branch banks can no longer provide this service. Similarly, the branch banks can be constructed to provide alternate service modes, such as the buffering of check requests in case of failure at the money-center bank, or buffering of low-priority check requests in case of failure of the full-bandwidth communications link.

Dealing with particular faults is only a small part of the problem. In practice, it will be necessary to deal with fault *sequences*, e.g., the loss of a communications link when the system has already experienced the loss of a node. In a large infrastructure network application, there are so many components that faults arising in sequence are a distinct possibility merely on stochastic grounds. However, cascading failures, sequenced terrorist attacks, or coordinated security attacks all yield fault sequences with faults that are potentially related, e.g., all links from a node are lost in some sequence or all the nodes in a geographic region are lost.

The various circumstances of interest can be described using a finite-state machine where each state is associated with a particular fault sequence. As well as enumeration of the states and associated state transitions associated with the faults that can arise, it is necessary to specify what has to be done on entry to each state in order to continue to provide service. Thus, application-

related actions have to be defined for each state transition, and the actions have to be tailored to both the initial state and the final state of the transition. Wide-area power failure has to be handled very differently if it occurs in a benign state versus when it occurs following a traumatic loss of computing equipment perhaps associated with a terrorist attack.

For this simple three-node example, we constructed a prototype error recovery specification to explore some of the issues involved in implementing a fault-tolerant system. The specification can be found in Appendix A. The first part of the specification is a description of the system itself. There are two aspects of the application that are described: the system architecture and the functionality (or services) provided by the system components. The description of the system architecture (Section A.1) consists of a listing of nodes (including attached databases) and connections. The functionality of each system component (Section A.2) is a listing of different services provided by each component of the system architecture, including alternate services available in case of various system failures. The system description is obviously informal, but it provides a basis for specification of the various system states that arise in the event of faults.

The second part of the specification is a description of the finite-state machine (Section A.3) and associated transitions (Section A.4) for the system and the fault sequences that are of concern. The initial state of the finite-state machine consists of a list of the services that are operational in the case of a fully-functional application. Transitions from this initial state are caused by faults, which manifest themselves as failures in one or more of the operational services. A high-level description of the fault that causes each transition is contained in the state description. In the finite-state machine transition section, a list of response activities is associated with each transition to attempt reconfiguration for the continued provision of service, as well as a list of activities to be undertaken upon repair of the fault.

In this simple three-node example, there are eleven components outlined in the specification of the system architecture that can fail, which would lead to faults at the system level. There are twenty-four services (some alternate) that those faults can affect; in the completely-operational initial state, there are ten services necessary to effect the fully-functional financial payments system. From the initial state there are eight possible faults causing transitions to other states. From those eight single-fault states, there are seventy-nine possible states should another fault occur. The finite-state machine in Appendix A only describes two sequential faults for this system. The complete finite-state machine enumerating all the states associated with the various possible fault sequences would have hundreds of states, even for a simple application system such as this.

The difficulties in achieving survivability, even in a system as simple as this example application, are clear. The first challenge lies in describing the relevant parts of the application, the system architecture and the functionality. Then, both the initial configuration and the changes to the system configuration in terms of that system description must be specified; the problem with state explosion and the impracticality of attempting to describe the finite state machine for a large network application is immediately obvious from the complications in this trivial system.

5 Solution Overview

As seen in the motivating example, to make a critical application fault tolerant, it is necessary to introduce mechanisms to recognize the errors of interest, maintain state information about the system to the extent that it affects error recovery, and define the required error recovery from all possible system states. The size of current and expected critical information systems, the variety and

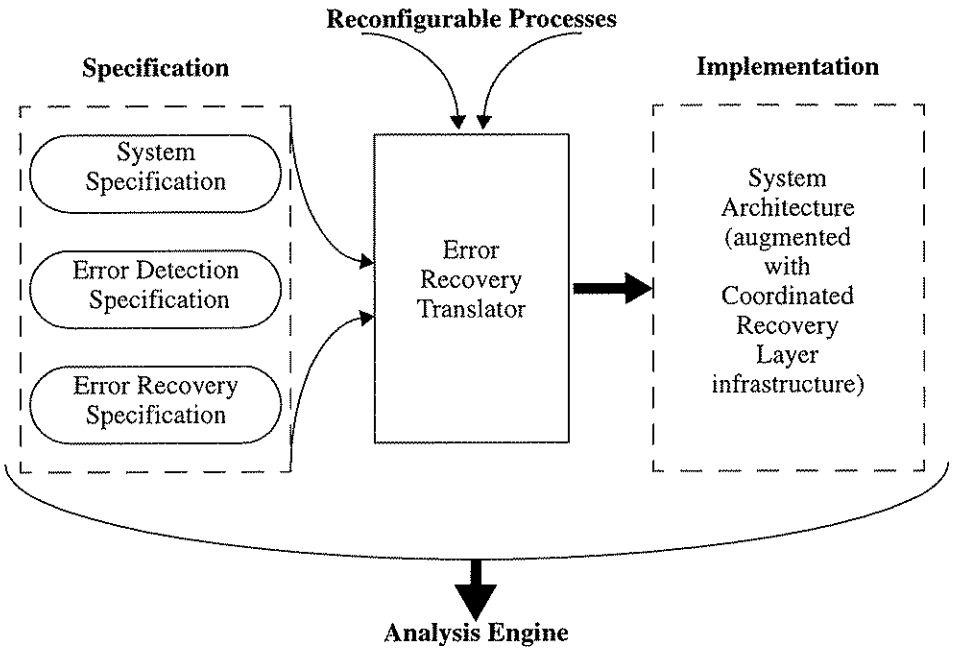

Figure 3. Error recovery methodology.

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. Programming fault tolerance for such a system using conventional methods is quite impractical.

For these reasons, our solution approach is based on the use of a formal specification to describe the required error detection and error recovery, and the use of synthesis to generate the implementation from the formal specification. A formal specification describes the system, the faults that must be tolerated, and the application responses to those faults. The use of a formal specification notation enables a translator, the Error Recovery Translator, to synthesize portions of the implementation dealing with error recovery. In addition to the specification, input to the translator includes a special type of process—called a reconfigurable process—that supports the application reconfigurations described in the specification. The system architecture will consist of these reconfigurable processes, the synthesized code produced by the translator, and a support infrastructure called the Coordinated Recovery Layer that provides services to the reconfigurable processes. Finally, an analysis engine provides various static and run-time checking capabilities to ensure correctness of the error recovery activities.

An overview of our solution approach is shown in Figure 3. The major solution components and issues relating to each aspect of the solution are described in subsequent sections of this paper:

- Specification and Synthesis
  In Section 6, we explore the issues involved in specifying and synthesizing error recovery. The specification is broken down into a variety of parts that define the different aspects of the problem as described previously. A unique element of the approach, however, is the use of a *multi-level* specification: the required error detection and error recovery specifications are written in terms of high-level abstract entities, defined in an intermediate specification that exports high-level entities but is itself written in terms of the low-level system components. The use of multiple specification levels for abstraction enables better control over the size and scope of these specifications. In addition, a synthesizer to process these formal specification notations has been built; the grammar for this Error Recovery Translator and the issues involved in generating code are presented in this section.
- Implementation: Node and System Architecture
  In Section 7, we explore the issues involved in an implementation architecture to support error recovery. At the node level, each node must be constructed such that it supports reconfiguration and incorporates the generated code that implements reconfiguration. At the system level, coordination and control services must be provided to the reconfiguring nodes; a global entity called the Coordinated Recovery Layer provides these support services.
- Analysis
  In Section 8, we explore the issues involved in analysis of the error recovery specifications and implementation. The use of formal specification permits various forms of analyses to check the error recovery algorithms: various forms of syntactic and semantic analyses are enabled by the formal specification notations, including invariant assertion checking. In addition, the implementation architecture supports various run-time checks to be performed.

6 Issues in Specification and Synthesis of Error Recovery

The first issue of concern in formally specifying error recovery is determining the components required in a specification language. Using the simple three-node system described in the motivating example, it can be seen that a finite-state machine must be constructed to specify the initial configuration of the system and any reconfigurations required to recover from system errors. This finite-state machine consists of all possible states the system could be in given the system errors that are of interest and should be handled, and the activities to undertake on transition from one system state to another. Two key aspects of the system must be described: the system architecture and the services that the nodes of the system provide.

As seen in the motivating example, even for a simple system an error recovery specification can become very large and unwieldy. Three observations can be made to deal with the specification size and state explosion problems:

- The specification notation must consist of multiple sub-specifications for describing the various components of the error recovery solution, e.g. the relevant system characteristics and the finite-state machine. These sub-specifications must be integrated to describe the overall error recovery solution, but the use of multiple sub-specifications enables different notations to be utilized and optimized for the particular aspect of the solution being addressed.
- The specification notation must be enhanced to accommodate larger numbers of nodes. One

way of achieving this would be to introduce and integrate some form of set-based notation to enable description and manipulation of large numbers of nodes simultaneously.

- The specification itself must be constructed in such a way as to keep it manageable: for example, portions of the system can be abstracted and consolidated into single objects in the specification, thus ensuring that the specification deals with and manipulates small numbers of objects regardless of how many actual nodes there are in the system.

Using the observations from the exercise in the motivating example, the first-generation RAPTOR specification notation was devised to specify error recovery in critical information systems.

6.1 The RAPTOR Specification Notation

The first-generation RAPTOR specification notation involves four major sub-specifications:

- Error Detection Specification (EDS)
  The error-detection specification defines the overall systems states that are associated with the various faults of interest.
- Error Recovery Specification (ERS)
  The error-recovery specification defines the necessary state changes from any acceptable system reconfiguration to any other in terms of topology, functionality, and geometry (assignment of services to nodes).
- System Architecture Specification (SAS)
  The system architecture specification describes the topology of the system and platform including the computing nodes, the communications links, and detailed parametric information for key characteristics. For example, nodes are named and described additionally with node type, hardware details, operating system, software versions, and so on. Links are specified with connection type and bandwidth capabilities.
- Service-Platform Mapping Specification (SPMS)
  The service-platform mapping specification relates the names of programs to the node names described in the SAS. The program descriptions in the SPMS include the services that each program provides, including alternate and degraded service modes.

Given the number of states that a large distributed system could enter as a result of the manifestation of a sequence of faults, it is clear that some form of accumulation of states or other simplification must be imposed if an approach even to specification of fault tolerance is to be tractable. The key to this simplification lies in the fact that many nodes in large networks, even those providing critical infrastructure service, do not need to be distinguished for purposes of fault tolerance. In the banking application, for example, it is clear that the loss of computing service at any single branch is both largely insignificant and largely independent of which branch is involved. Conversely, the loss of even one of the main Federal Reserve computing or communications centers would impede the financial system dramatically—some nodes are much more critical than others. However, the loss of 10,000 branch banks (for example because of a common-mode software error) would be extremely serious—even non-critical nodes have an impact if sufficient of them are lost at the same time.

To cope with the required accumulation of states, the overall specification is made two-level,

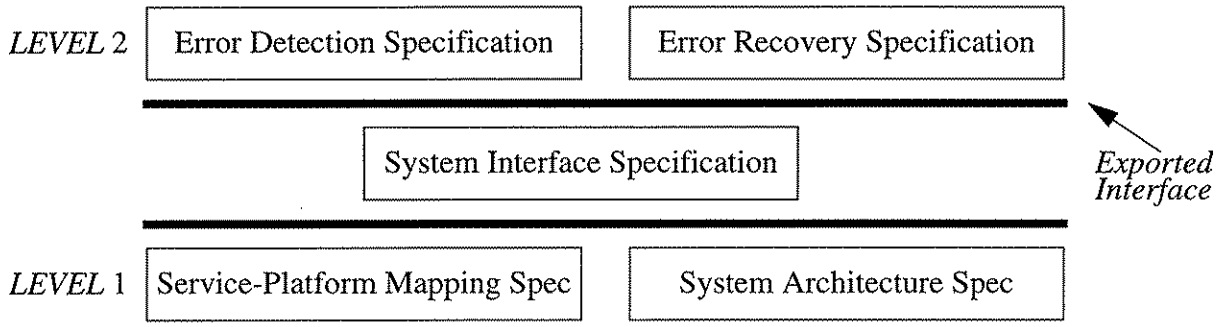

Figure 4. Two-level specification structure.

and we add a fifth element to the specification approach. The SAS and the SPMS are declarative specifications, and in practice the content of these specifications are databases of facts about the system architecture and configuration. The EDS and ERS are both algorithmic specifications—they describe algorithms that have to be executed to perform error detection and recovery respectively. In principle, these algorithms can be written using the information contained in the SAS and SPMS. But it is precisely this approach that leads to the state explosion in specification. The SAS and SPMS describing these systems are just too big.

The fifth element is the *system interface specification* (SIS). This is a specification that defines major system objects in terms of the lower-level entities contained in the SAS and SPMS. These objects are exported from the SAS and SPMS and become the objects with which the EDS and ERS are written. This structure is shown in Figure 4.

The overall structure of the ERS is that of a (traditional) finite-state machine that characterizes fault conditions as states (defined using sets) and the requisite responses to each fault are associated with state transitions. The fault conditions of concern for a given system are declared and described in the EDS. Arcs in the ERS finite-state machine are labeled with these fault conditions and show the state transitions for each fault from every relevant state. The actions associated with any given transition are in the ERS and are extensive because each action is essentially a high-level program that implements the error-recovery component of the full system survivability specification. The complete system-survivability specification documents the different states (system environments) that the system can be in, including the errors that will be detected and handled. In the RAPTOR specification notation's ERS a notational construct was designed to describe the finite-state machine of the system through all system errors. The notational construct for the finite-state machine enables brute-force description of all possible relevant failures in all possible states as well as the responses to those failures.

In summary, a first-generation RAPTOR specification consists of five subsections that correspond to the five sub-specifications outlined above. An example fragment of a RAPTOR specification is shown in Figure 5. This example is incomplete and uses comments for simplicity but it illustrates some of the material needed to define a wide-area coordinated security attack on the banking system and a hypothetical response that might be required.

```
RAPTOR-SAS:                    -- System architecture specification
   -- Use Z-like given sets for illustration
   -- [retail_banks], [regional_banks], [federal_reserve]

RAPTOR-PMS:                    -- System platform mapping specification
   forall i: retail_banks[i]       -> customer_service;
                                      local_payment;
   forall i: regional_banks[i]     -> customer_account_management;
                                      regional_payment;
   forall i: federal_reserve[i]    -> member_bank_account_management;
                                      national_payment;

RAPTOR-SIS:                    -- System interface specification
   Events:
      -- attack       == intrustion_detection_alarm triggered;
   Objects:
      branch_banks    == {i : bank | i memberof retail_banks}
      district_banks  == {i : bank | i memberof regional_banks}
      central_banks   == {i : bank | i memberof federal_reserve}

RAPTOR-EDS:                    -- Error detection specification
      coordinated_attack == card(attack(branch_banks)) > 1000
                            OR card(attack(district_banks)) > 3
                            OR card(attack(central_banks)) > 1;
      -- define this attack to be more than 10 branches or
      -- more than 3 money center or more than one Federal reserve
      -- bank detects an intrusion (via an intrusion detection system)

RAPTOR-ERS:                    -- Error recovery specification
   On coordinated_attack:
      branch_banks         -> customer_service.terminate;
                              local_payment.terminate;
                              local_enquiry.start;
      money_center_banks   -> customer_account_management.terminate;
                              regional_payment.terminate;
                              commercial_account_management.start;
      federal_reserve      -> member_bank_account_management.terminate;
                              national_payment.limit;
```

Figure 5. Skeleton RAPTOR specification.

6.2 RAPTOR Parser: Error Recovery Translator

The first-generation RAPTOR specification language has a parser, constructed using LEX and YACC. The parser processes all five sub-specification notations and generates C++ code for the actuators of the nodes in the run-time system. This Error Recovery Translator is constructed from a grammar with 41 productions and 31 tokens (presented in Appendix B). It contains facilities for simple set enumeration and composition, boolean logic, and quantifiers. The translator generates code on a per-node basis for all nodes declared in the System Architecture Specification.

The grammar for the RAPTOR language parses the RAPTOR sub-specification languages in a

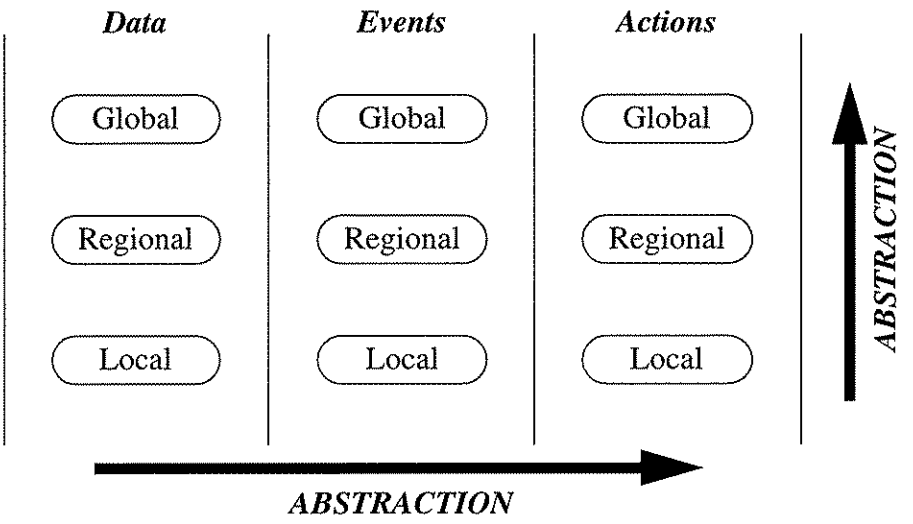

Figure 6. Two directions of abstraction.

6.4 Concepts for the Second-Generation RAPTOR Specification Notation

The consideration of issues in the finite-state machine description and state explosion problem in these critical, large-scale systems points towards the exploration of abstraction and hierarchy to help control state. These systems are large and geographically distributed; thus, there will be a notion of regional control and recovery. This characteristic lends itself to hierarchical control of error detection and recovery. This introduces abstraction in a complementary manner, or direction, to the abstraction discussed previously. In the previous form of abstraction, information or data is abstracted at lower levels of the system, then aggregated into sets (objects) at the system level and events (system errors) occurring with respect to those objects; finally, those events cause actions to occur in response, typically reconfiguration to recover from those system errors.

Hierarchy introduces abstraction in a second direction though (see Figure 6). Data, events, and actions can be occurring independently at different levels of the system hierarchy, such as at the local level, regional level, and global level. It is then possible for data, events, and actions at the local level to be passed up to the regional level, and so on to the global level.

This second abstraction mechanism can be used to help control the state explosion problem at the system level. It should be possible to construct finite-state machines at the node or local level to control error detection and error recovery internally, and only pass data, events, or actions up to the regional level in circumstances where the situation calls for that. Likewise, the system or global level state machine can be simplified by handling regional data, events, and actions at the lower level whenever possible.

7 Issues in a System Architecture for Error Recovery

A critical information system that supports error recovery in the manner described above must

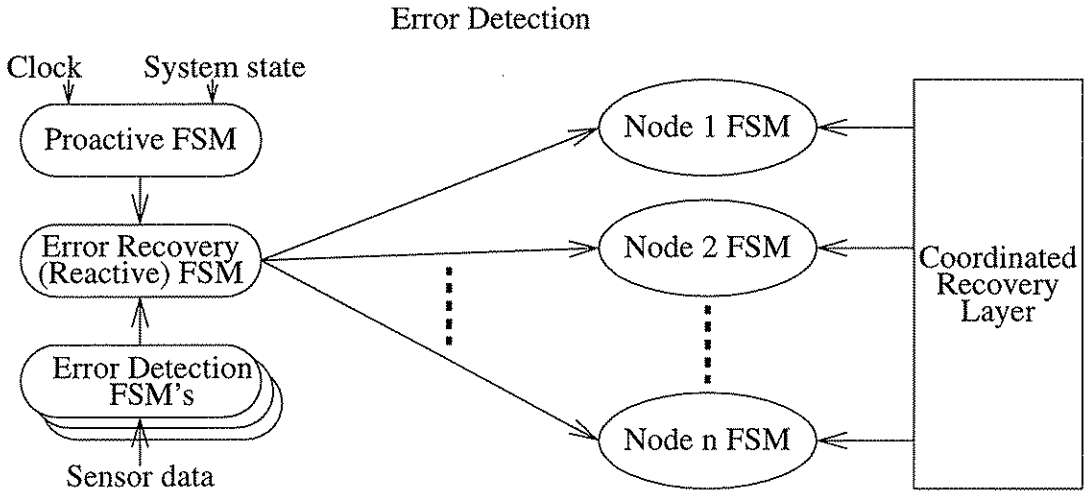

Figure 7. System architecture for error recovery.

have architectural support. In particular, both at the node level and the system level there must be mechanisms in place to effect reconfiguration. At the node level, the synthesized code for the actuator generated by the Error Recovery Translator must be integrated with the existing application code. At the system level, support for synchronization and coordination must be provided for reconfiguring nodes. The Coordinated Recovery Layer is a system-level construct that provides these services to the nodes.

7.1 Observations and Analysis

In terms of the implementation of the system architecture, a key design question concerns the location of the services provided by the Coordinated Recovery Layer. For example, one important coordination service is coordinated commitment by all nodes regarding reconfiguration actions: all of the nodes must receive the reconfiguration command and commit to that action at the same time.

One possible mechanism for implementing coordinated commitment is the two-phase commit protocol. Clearly, the nodes will have to participate in this protocol, so the services that are provided to the nodes by the Coordinated Recovery Layer really must exist within the node applications. However, the command to reconfigure will come from a separate entity in the survivability architecture, the control system. Therefore, one possible implementation would be to make the control system the coordinator in the two-phase commit protocol, thus locating that particular functionality of the Coordinated Recovery Layer in the control system. A key design question is whether all such services of the Coordinated Recovery Layer will exist in the control system and nodes (thus reducing the notion of the Coordinated Recovery Layer to a set of services provided in a library), or are there other services that must exist in a separate entity in the system that will be called the Coordinated Recovery Layer.

7.2 Future Directions

The second-generation RAPTOR specification notation points towards an implementation architecture similar to that pictured in Figure 7. The introduction of hierarchic finite-state machines for events, data, and actions requires finite-state machines at each node to control local error recovery. Sensor data from the various nodes will still feed into an error detection component to determine regional and global errors, and any faults recognized at those levels will initiate transitions at the higher-level error recovery finite-state machine(s). Finally, a finite-state machine for proactive error recovery activities must be introduced to control operations such as coordinated checkpointing.

8 Issues in Analysis of Error Recovery

The use of a formal notation for specification of error recovery permits various forms of analysis to be performed for correctness of the algorithms. The use of a parser checks the syntax of the specifications automatically to avoid simple errors of form. In addition, tools exist to analyze finite state machines for such characteristics as states with no exiting transitions.

More importantly though, semantic analysis can be performed on the specifications to ensure correctness of the error recovery. For example, the error recovery specification can be augmented with assertions to denote the fundamental processing or service to be provided during each state. Then the specification can be analyzed to prove that the required level of service is provided irrespective of the sequence of faults.

The system architecture also permits analysis of the run-time system. The control system continually monitors various nodes; the state information being collected from these nodes can include run-time assertions that check the provision of various levels of service.

9 Related Work

9.1 Fault-Tolerant Systems

Many system-level approaches exist that provide various subsets of abstractions and services. In this subsection we survey some of the existing work on fault-tolerant system architectures.

9.1.1. Cristian/Advanced Automation System. Cristian provided a survey of the issues involved in providing fault-tolerant distributed systems [8]. He presented two requirements for a fault-tolerant system: 1) mask failures when possible, and 2) ensure clearly specified failure semantics when masking is not possible. The majority of his work, however, dealt with the masking of failures.

An instantiation of Cristian's fault tolerance concepts was used in the replacement Air Traffic Control (ATC) system, called the Advanced Automation System (AAS). The AAS utilized Cristian's fault-tolerant architecture [9]. Cristian described the primary requirement of the air traffic control system as ultra-high availability and stated that the approach taken was to design a system that can automatically mask multiple concurrent component failures.

The air traffic control system described by Cristian handled relatively low-level failures: redundancy of components was utilized and managed in order to mask these faults. Cristian structured the fault-tolerant architecture using a "depends-on" hierarchy, and modelled the system in

terms of servers, services, and a "uses" relation. Redundancy was used to mask both hardware and software failures at the highest level of abstraction, the application level. Redundancy was managed by application software server groups [9].

9.1.2. Birman/ISIS, Horus, and Ensemble. A work similar to that of Cristian is the "process-group-based computing model" presented by Birman. Birman introduced a toolkit called ISIS that contained system support for process group membership, communication, and synchronization. ISIS balanced trade-off's in closely synchronized distributed execution (which offers easy understanding) and asynchronous execution (which achieves better performance through pipelined communication) by providing the virtual synchrony approach to group communication. ISIS facilitated group-based programming by providing a software infrastructure to support process group abstractions. Both Birman and Cristian's work addressed a "process-group-based computing model," though Cristian's AAS also provided strong real-time guarantees made possible by an environment with strict timing properties [5].

Work on ISIS proceeded in subsequent years resulting in another group communications system, Horus. The primary benefit of Horus over ISIS was a flexible communications architecture that can be varied at runtime to match the changing requirements of the application and environment. Horus achieved this flexibility using a layered protocol architecture in which each module is responsible for a particular service [35]. Horus also worked with a system called Electra, which provided a CORBA-compliant interface to the process group abstraction in Horus [21]. Another system that built on top of Electra and Horus together, Piranha, provided high availability by supporting application monitoring and management facilities [22].

Horus was succeeded by a new tool for building adaptive distributed programs, Ensemble. Ensemble further enabled application adaptation through a stackable protocol architecture as well as system support for protocol switching. Performance improvements were also provided in Ensemble through protocol optimization and code transformations [36].

An interesting note on ISIS, Horus, and Ensemble was that all three acknowledged the security threats to the process group architecture and each incorporated a security architecture into its system [29], [30], [31].

9.1.3. Other system-level approaches. Another example of fault tolerance that focuses on communication abstractions is the work of Schlichting *et al.* The result of this work is a system called Coyote that supports configurable communication protocol stacks. The goals are similar to that of Horus and Ensemble, but Coyote generalizes the composition of microprotocol modules allowing non-hierarchical composition (Horus and Ensemble only support hierarchical composition). In addition, Horus and Ensemble are focusing primarily on group communication services while Coyote supports a variety of high-level network protocols [4].

Many of the systems mentioned above focus on communication infrastructure and protocols for providing fault tolerance; another approach focuses on transactions in distributed systems as the primary primitive for providing fault tolerance. One of the early systems supporting transactions was Argus, developed at MIT. Argus was a programming language and support system that defined transactions on software modules, ensuring persistence and recoverability [6].

Another transaction-based system, Arjuna, was developed at the University of Newcastle upon Tyne. Arjuna is an object-oriented programming system that provides atomic actions on objects using C++ classes [32]. The atomic actions ensure that all operations support the properties of serializability, failure atomicity, and permanence of effect.

9.1.4. Discussion. A common thread through the approach taken to fault tolerance in all of these systems is that faults are masked; each of these systems attempts to provide transparent masking of failures when a fault arises. Masking requires redundancy, and not all faults can be masked because it is not possible to build enough redundancy into a system to accommodate all faults in that way. Therefore, there will be a class of faults that these approaches to fault tolerance cannot handle because there is insufficient redundancy to tolerate them by masking.

Another interesting note is that these approaches tend to be communication-oriented. This is easily understandable - fault tolerance in general is dependent on redundancy, and one key to managing redundancy is maintaining consistent views of the state across all redundant entities. Supporting such a requirement within the communications framework—building guarantees into that framework—is a common approach to providing fault tolerance, but communications is not the only aspect of the system that must be addressed for a comprehensive error recovery strategy.

Finally, the scale of these systems tends not to be on the order of critical information systems.

9.2 Fault Tolerance in Wide-area Network Systems

Fault tolerance is typically applied to relatively small-scale systems, dealing with single processor failures and limited redundancy. Critical information systems are many orders of magnitude larger than the distributed systems that most of the previous work has addressed. There are, however, a few research efforts addressing fault tolerance in large-scale, wide-area network systems.

In the WAFT project, Marzullo and Alvisi are concerned with the construction of fault-tolerant applications in wide-area networks. Experimental work has been done on the Nile system, a distributed computing solution for a high-energy physics project. The primary goal of the WAFT project is to adapt replication strategies for large-scale distributed applications with dynamic (unpredictable) communication properties and a requirement to withstand security attacks. Nile was implemented on top of CORBA in C++ and Java. The thrust of the work thus far is that active replication is too expensive and often unnecessary for these wide-area network applications; Marzullo and Alvisi are looking to provide support for passive replication in a toolkit [1].

The Eternal system, developed by Melliar-Smith and Moser, is middleware that operates in a CORBA environment, below a CORBA ORB but on top of their Totem group communication system. The primary goal is to provide transparent fault tolerance to users [25].

Babaoglu and Schiper are addressing problems with scaling of conventional group technology. Their approach for providing fault tolerance in large-scale distributed systems consists of distinguishing between different roles or levels for group membership and providing different service guarantees to each level [3].

9.2.1. Discussion. While the approaches discussed in this section accommodate systems of a larger scale, many of the concerns raised previously still apply. These efforts still attempt to mask faults using redundancy and they are still primarily communications-oriented. There is still a class of faults that cannot be handled because there is insufficient redundancy.

9.3 Reconfigurable Distributed Systems

Given the body of literature on fault tolerance and the different services being provided at each abstraction layer, many types of faults can be handled. However, the most serious fault—the catastrophic, non-maskable fault—is not addressed by the previous related work. The previous

approaches rely on having sufficient redundancy to cope with the fault and mask it; there are always going to be classes of faults for which this is not possible. For these faults, reconfiguration of the existing services on the remaining platform is required.

Considerable work has been done on reconfigurable distributed systems. Some of the work deals with reconfiguration for the purposes of evolution, as in the CONIC system, and, while this work is relevant, it is not directly applicable because it is concerned with reconfiguration that derives from the need to upgrade rather than cope with major faults. Less work has been done on reconfiguration for the purposes of fault tolerance. Both types of research are discussed in this section.

9.3.1. Reconfiguration supporting system evolution. The initial context of the work by Kramer and Magee was dynamic configuration for distributed systems, incrementally integrating and upgrading components for system evolution. CONIC, a language and distributed support system, was developed to support dynamic configuration. The language enabled specification of system configuration as well as change specifications, then the support system provided configuration tools to build the system and manage the configuration [18].

More recently, they have modelled a distributed system in terms of processes and connections, each process abstracted down to a state machine and passing messages to other processes (nodes) using the connections. One relevant finding of this work is that components must migrate to a "quiescent state" before reconfiguration to ensure consistency through the reconfiguration; basically, a quiescent state entailed not being involved in any transactions. The focus remained on the incremental changes to a distributed system configuration for evolutionary purposes [19].

The successor to CONIC, Darwin, is a configuration language that separates program structure from algorithmic behavior [24]. 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 [23].

9.3.2. Reconfiguration supporting fault tolerance. Purtilo developed the Polylith Software Bus, a software interconnection system that provides a module interconnection language and interfacing facilities (software toolbus). Basically, Polylith encapsulates all of the interfacing details for an application, where all software components communicate with each other through the interfaces provided by the Polylith software bus [28].

Hofmeister extended Purtilo's work by building additional primitives into Polylith for support of reconfigurable applications. Hofmeister studied the types of reconfigurations that are possible within applications and the requirements for supporting reconfiguration. Hofmeister leveraged heavily off of Polylith's interfacing and message-passing facilities in order to ensure state consistency during reconfiguration [13].

Welch and Purtilo have extended Hofmeister's work in a particular application domain, Distributed Virtual Environments. They utilized Polylith and its reconfiguration extensions in a toolkit that helps to guide the programmer in deciding on proper reconfigurations and implementations for these simulation applications [37].

9.3.3. Discussion. The research on reconfiguration for the purposes of evolution is interesting but of course does not work on the same time scale as required for survivability in critical information systems. Critical information systems have performance requirements that must still be met by an

error recovery mechanism; reconfiguration during evolution is not concerned with performance in general.

Reconfiguration for the purposes of fault tolerance, however, does concern itself with performance. However, the existing solutions do not accommodate systems on the scale of critical information systems. One might argue as well that these research efforts do not handle the complexity and heterogeneity of critical information systems.

10 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 the problem 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 two-level structure. 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 network applications in a manageable way. The price of achieving this is a loss in flexibility. The utility of the approach is presently under investigation as are the details of implementation.

11 Acknowledgments

This effort sponsored 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.

12 References

[1] Alvisi, L. and K. Marzullo. "WAFT: 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] Babaoglu, O. and A. Schiper. "On Group Communication in Large-Scale Distributed Systems," ACM Operating Systems Review, Vol. 29 No. 1, January 1995, pp. 62-67.

[4] 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.

[5] 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.

[6] Birman, K. Building Secure and Reliable Network Applications. Manning, Greenwich, CT, 1996.

[7] 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.

[8] Cristian, F. "Understanding Fault-Tolerant Distributed Systems," Communications of the ACM, Vol. 34 No. 2, February 1991, pp. 56-78.

[9] 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.

[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 '98, 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]Knight, J. and K. Sullivan. "Towards a Definition of Survivability," Proceedings of the Third Information Survivability Workshop, October 2000.

[18]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.

[19]Kramer, J. and J. Magee. "The Evolving Philosophers Problem: Dynamic Change Management," IEEE Transactions on Software Engineering, Vol. 16 No. 11, November 1990, pp. 1293-1306.

[20]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.

[21]Maffeis, S. "Electra - Making Distributed Programs Object-Oriented," Technical Report 93-

17, Department of Computer Science, University of Zurich, April 1993.

[22]Maffeis, S. "Piranha: A CORBA Tool For High Availability," IEEE Computer, Vol. 30 No. 4, April 1997, pp. 59-66.

[23]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.

[24]Magee, J. and J. Kramer. "Darwin: An Architectural Description Language," http://www-dse.doc.ic.ac.uk/research/darwin/darwin.html, 1998.

[25]Melliar-Smith, P. and L. Moser. "Surviving Network Partitioning," IEEE Computer, Vol. 31 No. 3, March 1998, pp. 62-68.

[26]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.

[27]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.

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

[29]Reiter, M., K. Birman, and L. Gong. "Integrating Security in a Group Oriented Distributed System," Proceedings of the 1992 IEEE Symposium on Research in Security and Privacy, IEEE Computer Society Press, Los Alamitos, CA, May 1992, pp. 18-32.

[30]Reiter, M., K. Birman, and R. van Renesse. "A Security Architecture for Fault-Tolerant Systems," ACM Transactions on Computer Systems, Vol. 12 No. 4, November 1994, pp. 340-371.

[31]Rodeh, O., K. Birman, M. Hayden, Z. Xiao, and D. Dolev. "Ensemble Security," Technical Report TR98-1703, Department of Computer Science, Cornell University, September 1998.

[32]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.

[33]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.

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

[35]van Renesse, R., K. Birman, and S. Maffeis. "Horus: A Flexible Group Communications System," Communications of the ACM, Vol. 39 No. 4, April 1996, pp. 76-83.

[36]van Renesse, R., K. Birman, M. Hayden, A. Vaysburd, and D. Karr. "Building Adaptive Systems Using Ensemble," Technical Report TR97-1638, Department of Computer Science, Cornell University, July 1997.

[37]Welch, D. "Building Self-Reconfiguring Distributed Systems using Compensating Reconfiguration," Proceedings of the 4th International Conference on Configurable Distributed Systems, IEEE Computer Society Press, Los Alamitos, CA, May 1998.

Appendix A
3-Node Example Prototype System Specification

This example system is a small-scale financial payments application, effecting value transfer between customer accounts. The system consists of three nodes, two branch banks and one money-center bank, and various connections between these nodes. The branch banks are intended to provide customer access (check deposit facilities) and local information storage (customer accounts), while the money-center bank is intended to track branch bank asset balances and route checks for clearance between the two branch banks.

A.1 System Architecture Specification

The system architecture is pictured in Figure 8.

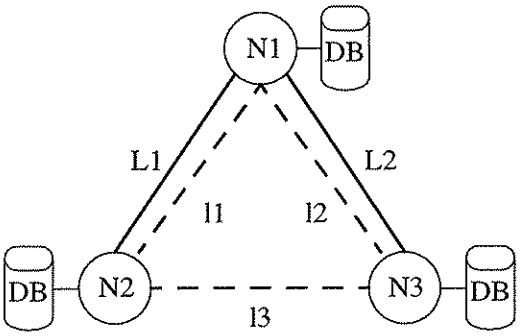

Figure 8. Example financial payments application.

The system consists of a 3 nodes:

- N1: money-center bank (MCB)
- N2: a branch bank (BB)
- N3: branch bank (BB)

There is a database (DB) attached to each node:

- DB(N1)
- DB(N2)
- DB(N3)

There are five total connections (primary and backup) between the various nodes:

- L1: full-bandwidth link between N1 and N2
- L2: full-bandwidth link between N1 and N3
- l1: low-bandwidth backup link between N1 and N2
- l2: low-bandwidth backup link between N1 and N3
- l3: full-bandwidth backup link between the two branch banks, N2 and N3

A.2 System Functionality Specification

In this section, the services that each node provides will be specified (named), with a brief description of each service.

The money center bank N1 provides the following services:

- MCB1: Route requests
- MCB2: Maintain branch total balances (DB service)
- MCB3: Buffer requests for a branch bank (Alternate)
- MCB4: Send buffered requests (Alternate)
- MCB5: Accept requests from customers (Alternate)

The branch banks N2 and N3 provide the following services:

- BB1: Accept requests from customers, routing to other branch bank if necessary
- BB2: Accept requests from other branch banks
- BB3: Process requests, maintaining customer balances (DB service)
- BB4: Buffer requests to pass up (Alternate)
- BB5: Send buffered requests (Alternate)
- BB6: Send high-priority requests, queue others (Alternate)
- BB7: Pass requests directly to branch bank (Alternate)

The two full-bandwidth links (L1 and L2) provide the following service:

- FC1: Pass full bandwidth data over full connection

The two low-bandwidth backup links (l1 and l2) provide the following service:

- DC1: Pass limited bandwidth data over degraded connection

The full-bandwidth backup link (l3) provides the following service:

- AC1: Pass full bandwidth data over alternate connection

A.3 Finite-State Machine Specification

S0: Initial state
N1: MCB1, MCB2
N2: BB1, BB2, BB3
N3: BB1, BB2, BB3
L1: FC1
L2: FC1

The first level of transitions from the initial state consist of a single fault occurring from the initial state. There are eight such transitions leading to the following eight states: S1, S2, S3, S4, S5, S6, S7, S8. The services after the transition include any alternate services started as a result of application reconfiguration.

S1: Process failure - MCB1 (N1)
N1: MCB2
N2: BB1, BB2, BB3, BB4
N3: BB1, BB2, BB3, BB4
L1: FC1
L2: FC1

S2: Database failure - MCB2 (N1)
N1: MCB1, MCB3
N2: BB1, BB2, BB3
N3: BB1, BB2, BB3
L1: FC1
L2: FC1

S3: Full node failure - MCB1, MCB2 (N1)
N1: -
N2: BB1, BB2, BB3, BB7

N3: BB1, BB2, BB3, BB7
L1: FC1
L2: FC1
I3: AC1

S4: Process failure - BB1 (either N2 or N3)
N1: MCB1, MCB2
N2: BB2, BB3
N3: BB1, BB2, BB3
L1: FC1
L2: FC1

S5: Process failure - BB2 (either N2 or N3)
N1: MCB1, MCB2, MCB3
N2: BB1, BB3
N3: BB1, BB2, BB3
L1: FC1
L2: FC1

S6: Database failure - BB3 (either N2 or N3)
N1: MCB1, MCB2, MCB3
N2: BB1, BB2, BB4
N3: BB1, BB2, BB3
L1: FC1
L2: FC1

S7: Full node failure - BB1, BB2, BB3 (either N2 or N3)
N1: MCB1, MCB2, MCB3
N2: -
N3: BB1, BB2, BB3
L1: FC1
L2: FC1

S8: Link failure - FC1 (either L1 or L2)
N1: MCB1, MCB2
N2: BB1, BB2, BB3, BB6
N3: BB1, BB2, BB3, BB6
L1: -
L2: FC1
I1: DC1

To handle a second sequential fault from the initial state, it is necessary to specify the list of faults that can occur from each of the above states. Then, each fault would necessitate another transition in the finite-state machine to a different state. The following are the second level of states in the finite-state machine and the fault that causes the transition to each state:

From State S1:

S10: Database failure - MCB2 (N1)
S11: Full node failure - MCB2 (N1)
S12: Process failure - BB1 (either N2 or N3)
S13: Process failure - BB2 (either N2 or N3)
S14: Database failure - BB3 (either N2 or N3)
S15: Process failure - BB4 (either N2 or N3)
S16: Full node failure - BB1, BB2, BB3, BB4 (either N2 or N3)
S17: Link failure - FC1 (either L1 or L2)

From State S2:
- S20: Database failure - MCB2 (N1)
- S21: Process failure - MCB3 (N1)
- S22: Full node failure - MCB2, MCB3 (N1)
- S23: Process failure - BB1 (either N2 or N3)
- S24: Process failure - BB2 (either N2 or N3)
- S25: Database failure - BB3 (either N2 or N3)
- S26: Full node failure - BB1, BB2, BB3 (either N2 or N3)
- S27: Link failure - FC1 (either L1 or L2)

From State S3:
- S30: Process failure - BB1 (either N2 or N3)
- S31: Process failure - BB2 (either N2 or N3)
- S32: Database failure - BB3 (either N2 or N3)
- S33: Process failure - BB7 (either N2 or N3)
- S34: Full node failure - BB1, BB2, BB3, BB7 (either N2 or N3)
- S35: Link failure - FC1 (either L1 or L2)
- S36: Link failure - AC1 (l3)

From State S4:
- S40: Process failure - MCB1 (N1)
- S41: Database failure - MCB2 (N1)
- S42: Full node failure - MCB1, MCB2 (N1)
- S43: Process failure - BB2 (same N2 or N3 as previous fault)
- S44: Database failure - BB3 (same N2 or N3 as previous fault)
- S45: Full node failure - BB2, BB3 (same N2 or N3 as previous fault)
- S46: Process failure - BB1 (different N2 or N3 from previous fault)
- S47: Process failure - BB2 (different N2 or N3 from previous fault)
- S48: Database failure - BB3 (different N2 or N3 from previous fault)
- S49: Full node failure - BB1, BB2, BB3 (different N2 or N3 from previous fault)
- S4a: Link failure - FC1 (either L1 or L2)
- S4b: Link failure - AC1 (l3)

From State S5:
- S50: Process failure - MCB1 (N1)
- S51: Database failure - MCB2 (N1)
- S52: Process failure - MCB3 (N1)
- S53: Full node failure - MCB1, MCB2, MCB3 (N1)
- S54: Process failure - BB1 (same N2 or N3 as previous fault)
- S55: Database failure - BB3 (same N2 or N3 as previous fault)
- S56: Full node failure - BB1, BB3 (same N2 or N3 as previous fault)
- S57: Process failure - BB1 (different N2 or N3 from previous fault)
- S58: Process failure - BB2 (different N2 or N3 from previous fault)
- S59: Database failure - BB3 (different N2 or N3 from previous fault)
- S5a: Full node failure - BB1, BB2, BB3 (different N2 or N3 from previous fault)
- S5b: Link failure - FC1 (either L1 or L2)

From State S6:
- S60: Process failure - MCB1 (N1)
- S61: Database failure - MCB2 (N1)
- S62: Process failure - MCB3 (N1)
- S63: Full node failure - MCB1, MCB2, MCB3 (N1)

S64: Process failure - BB1 (same N2 or N3 as previous fault)
S65: Database failure - BB2 (same N2 or N3 as previous fault)
S66: Process failure - BB4 (same N2 or N3 as previous fault)
S67: Full node failure - BB1, BB2, BB4 (same N2 or N3 as previous fault)
S68: Process failure - BB1 (different N2 or N3 from previous fault)
S69: Process failure - BB2 (different N2 or N3 from previous fault)
S6a: Database failure - BB3 (different N2 or N3 from previous fault)
S6b: Full node failure - BB1, BB2, BB3 (different N2 or N3 from previous fault)
S6c: Link failure - FC1 (either L1 or L2)

From State S7:

S70: Process failure - MCB1 (N1)
S71: Database failure - MCB2 (N1)
S72: Process failure - MCB3 (N1)
S73: Full node failure - MCB1, MCB2, MCB3 (N1)
S74: Process failure - BB1 (different N2 or N3 from previous fault)
S75: Process failure - BB2 (different N2 or N3 from previous fault)
S76: Database failure - BB3 (different N2 or N3 from previous fault)
S77: Full node failure - BB1, BB2, BB3 (different N2 or N3 from previous fault)
S78: Link failure - FC1 (either L1 or L2)

From State S8:

S80: Process failure - MCB1 (N1)
S81: Database failure - MCB2 (N1)
S82: Full node failure - MCB1, MCB2 (N1)
S83: Process failure - BB1 (either N2 or N3)
S84: Process failure - BB2 (either N2 or N3)
S85: Database failure - BB3 (either N2 or N3)
S86: Process failure - BB6 (either N2 or N3)
S87: Full node failure - BB1, BB2, BB3, BB6 (either N2 or N3)
S88: Link failure - DC1 (same l1 or l2 that was started because of previous fault)
S89: Link failure - FC1 (different L1 or L2 from previous fault)

A.4 Finite-State Machine Transitions

The transitions from the initial state to the next state caused by a single fault are described in this subsection. In the state descriptions above, the surviving services and the alternate services that were started are already specified. These actions describe the handling of the fault both in response to its occurrence and after the repair of that fault (to return to the initial state):

- Transition S0 to S1:
  Response: Start BB4 (both N2 and N3)
  Repair: Start BB5 (both N2 and N3)
  Stop BB4 (both N2 and N3)
  Stop BB5 (when each queue empty)
- Transition S0 to S2:
  Response: Start MCB3
  Repair: Start MCB4
  Stop MCB3
  Stop MCB4

- Transition S0 to S3:
  Response: Start AC1
  Start BB7 (both N2 and N3)
  Repair: Stop BB7 (both N2 and N3)
  Stop AC1
- Transition S0 to S4:
  (No response)
- Transition S0 to S5:
  Response: Start MCB3
  Repair: Start MCB4
  Stop MCB3
  Stop MCB4 (when queue empty)
- Transition S0 to S6:
  Response: Start MCB3
  Start BB4 (for node N2 or N3 with fault)
  Repair: Start MCB4
  Stop MCB3
  Stop MCB4 (when queue empty)
  Stop MCB (for repaired node N2 or N3)
- Transition S0 to S7:
  Response: Start MCB3
  Repair: Start MCB4
  Stop MCB3
  Stop MCB4 (when queue empty)
- Transition S0 to S8:
  Response: Start DC1 (for failed link L1 or L2)
  Start BB6
  Repair: Start BB5
  Stop BB6
  Stop BB5 (when queue empty)
  Stop DC1 (for repaired link L1 or L2)

Appendix B
YACC Grammar for the RAPTOR Specification Notation

The following is the YACC grammar for the first-generation RAPTOR Specification Notation.

```
%%

%token SAS
%token SPMS
%token SIS
%token EDS
%token ERS
%token NODE
%token <stringtype> NODE_NAME
%token TYPE
%token <stringtype> TYPE_NAME
%token PROP
%token <stringtype> PROP_NAME
%token EVENT
%token <stringtype> EVENT_NAME
%token SERVICE
%token <stringtype> SERVICE_NAME
%token SET
%token <stringtype> SET_NAME
%token <stringtype> SET_MEMBER
%token <stringtype> VAR
%token <stringtype> COMPONENT_TYPE
%token ERROR
%token <stringtype> ERROR_NAME
%token FAILURE
%token ARROW
%token FORALL
%token EXISTS
%token AND
%token OR
%token IN
%token <inttype> START
%token <inttype> STOP

%type <inttype> critical_service

%%

raptor_specification
      : sas spms sis eds ers
        { printf("Parsed a RAPTOR specification!!!\n"); }

sas
      : SAS declaration_list propositions

declaration_list
      : node_declarations
        { printf("CompletedNodes\n"); nodes.CompletedNodes(); }
      | declaration_list type_declarations
        { printf("CompletedTypes\n"); nodes.CompletedTypes(); }
      | declaration_list prop_declarations
```

```
        { printf("CompletedProps\n"); nodes.CompletedProps(); }

node_declarations
      : node_declaration
      | node_declarations node_declaration

node_declaration
      : NODE node_list

node_list
      : NODE_NAME
        { nodes.AddNode($1); }
      | node_list ',' NODE_NAME
        { nodes.AddNode($3); }

type_declarations
      : type_declaration
      | type_declarations type_declaration

type_declaration
      : TYPE TYPE_NAME
        { nodes.AddType($2); }

prop_declarations
      : prop_declaration
      | prop_declarations prop_declaration

prop_declaration
      : PROP PROP_NAME
        { nodes.AddProp($2); }

propositions
      : proposition
      | propositions proposition

proposition
      : TYPE_NAME '(' NODE_NAME ')' ';'
        { nodes.SetNodeType($3, $1); }
      | PROP_NAME '(' NODE_NAME ')' ';'
        { nodes.SetNodeProp($3, $1); }

spms
      : SPMS service_mappings
        { printf("CompletedMappings\n");
          nodes.CompletedNodeServiceMapping(); }

service_mappings
      : service_mapping
      | service_mappings service_mapping

service_mapping
      : single_node_service_mapping
      | multiple_node_service_mapping

single_node_service_mapping
      : NODE_NAME ARROW service_list ';'
        { printf("Mapping services to node\n");
```

```
          nodes.MapServicesToNode($1); }

multiple_node_service_mapping
      : FORALL TYPE_NAME ARROW service_list ';'
        { printf("Mapping services to type\n");
          nodes.MapServicesToType($2); }

service_list
      : SERVICE_NAME
        { nodes.AddService($1); }
      | service_list ',' SERVICE_NAME
        { nodes.AddService($3); }

sis
      : SIS set_declaration_list set_definitions

set_declaration_list
      : set_declarations
        { printf("CompletedSets\n");
          nodes.CompletedSets(); }

set_declarations
      : set_declaration
      | set_declarations set_declaration

set_declaration
      : SET SET_NAME
        { nodes.AddSet($2); }

set_definitions
      : set_definition
      | set_definitions set_definition

set_definition
      : SET_NAME '=' '{' set_list '}'
        { nodes.SetSet($1); }
      | SET_NAME '=' '{' set_iteration '}'
        { nodes.SetSet($1); }

set_list
      : SET_MEMBER
        { nodes.AddToTempSetList($1); }
      | set_list ',' SET_MEMBER
        { nodes.AddToTempSetList($3); }

set_iteration
      : VAR ':' COMPONENT_TYPE '|' PROP_NAME '(' VAR ')'
        { //printf("Vars = %s|%s, Type = %s, Prop = %s\n", $1, $7, $3, $5);
          nodes.AddPropToTempSet($5); }
      | VAR ':' COMPONENT_TYPE '|' TYPE_NAME '(' VAR ')'
        { //printf("Vars = %s|%s, Type = %s, Prop = %s\n", $1, $7, $3, $5);
          nodes.AddTypeToTempSet($5); }

eds
      : EDS error_declaration_list error_definitions

error_declaration_list
```

```
      : error_declarations
        { printf("CompletedErrors\n");
          nodes.CompletedErrors(); }

error_declarations
      : error_declaration
      | error_declarations error_declaration

error_declaration
      : ERROR ERROR_NAME
        { nodes.AddError($2); }

error_definitions
      : error_definition
      | error_definitions error_definition

error_definition
      : ERROR_NAME '=' conditions ';'

conditions
      : condition
      | '(' condition ')'
      | conditions conjunction condition
      | '(' conditions conjunction condition ')'

condition
      : FAILURE '(' NODE_NAME '.' SERVICE_NAME ')'
        { printf("Failure: Node = %s, Serv = %s\n", $3, $5); }
      | EVENT_NAME '(' NODE_NAME ')'
        { printf("Event = %s, Node = %s\n", $1, $3); }
      | '(' FORALL VAR IN SET_NAME '|' EVENT_NAME '(' VAR ')' ')'
        { printf("Var = %s|%s, Set = %s, Event = %s\n", $3, $9, $5, $7); }
      | '(' EXISTS VAR IN SET_NAME '|' EVENT_NAME '(' VAR ')' ')'
        { printf("Var = %s|%s, Set = %s, Event = %s\n", $3, $9, $5, $7); }

conjunction
      : AND
      | OR

ers
      : ERS error_activities
        { printf("CompletedResponses\n"); }

error_activities
      : per_error_activities
      | error_activities per_error_activities

per_error_activities
      : ERROR_NAME ':' per_node_activities
        { nodes.CompletedResponse($1); }

per_node_activities
      : per_node_responses
      | per_node_activities per_node_responses

per_node_responses
      : NODE_NAME ARROW response_list ';'
```

```
        { nodes.CompletedNodeResponse($1); }
      | SET_NAME ARROW response_list ';'
        { nodes.CompletedSetResponse($1); }

response_list
      : SERVICE_NAME '.' critical_service
        { nodes.AddResponse($1, $3); }
      | response_list ',' SERVICE_NAME '.' critical_service
        { nodes.AddResponse($3, $5); }

critical_service
      : START
        { $$ = CS_START; }
      | STOP
        { $$ = CS_STOP; }

%%
```

Appendix C
100-Node Banking Example

The following is the RAPTOR specification of error recovery for a 100-node banking system.

```
SYSTEM_ARCHITECTURE_SPECIFICATION


NODE frb1, frb2, frb3
NODE mcb100, mcb200, mcb300, mcb400, mcb500, mcb600, mcb700, mcb800, mcb900,
     mcb1000
NODE bb101, bb102, bb103, bb104, bb105, bb106, bb107, bb108, bb109
NODE bb201, bb202, bb203, bb204, bb205, bb206, bb207, bb208, bb209
NODE bb301, bb302, bb303, bb304, bb305, bb306, bb307, bb308, bb309
NODE bb401, bb402, bb403, bb404, bb405, bb406, bb407, bb408, bb409
NODE bb501, bb502, bb503, bb504, bb505, bb506, bb507, bb508, bb509
NODE bb601, bb602, bb603, bb604, bb605, bb606, bb607, bb608, bb609
NODE bb701, bb702, bb703, bb704, bb705, bb706, bb707, bb708, bb709
NODE bb801, bb802, bb803, bb804, bb805, bb806, bb807, bb808, bb809
NODE bb901, bb902, bb903, bb904, bb905, bb906, bb907, bb908, bb909
NODE bb1001, bb1002, bb1003, bb1004, bb1005, bb1006, bb1007, bb1008, bb1009

TYPE federal_reserve
TYPE money_center
TYPE branch

PROP east_coast
PROP north_east
PROP south_east
PROP north_central
PROP south_central
PROP north_west
PROP south_west
PROP west_coast

EVENT security_attack
EVENT node_failure
EVENT power_failure

federal_reserve(frb1); federal_reserve(frb2); federal_reserve(frb3);
money_center(mcb100);
branch(bb101); branch(bb102); branch(bb103);
branch(bb104); branch(bb105); branch(bb106);
branch(bb107); branch(bb108); branch(bb109);
money_center(mcb200);
branch(bb201); branch(bb202); branch(bb203);
branch(bb204); branch(bb205); branch(bb206);
branch(bb207); branch(bb208); branch(bb209);
money_center(mcb300);
branch(bb301); branch(bb302); branch(bb303);
branch(bb304); branch(bb305); branch(bb306);
branch(bb307); branch(bb308); branch(bb309);
money_center(mcb400);
branch(bb401); branch(bb402); branch(bb403);
branch(bb404); branch(bb405); branch(bb406);
branch(bb407); branch(bb408); branch(bb409);
```

```
money_center(mcb500);
branch(bb501); branch(bb502); branch(bb503);
branch(bb504); branch(bb505); branch(bb506);
branch(bb507); branch(bb508); branch(bb509);
money_center(mcb600);
branch(bb601); branch(bb602); branch(bb603);
branch(bb604); branch(bb605); branch(bb606);
branch(bb607); branch(bb608); branch(bb609);
money_center(mcb700);
branch(bb701); branch(bb702); branch(bb703);
branch(bb704); branch(bb705); branch(bb706);
branch(bb707); branch(bb708); branch(bb709);
money_center(mcb800);
branch(bb801); branch(bb802); branch(bb803);
branch(bb804); branch(bb805); branch(bb806);
branch(bb807); branch(bb808); branch(bb809);
money_center(mcb900);
branch(bb901); branch(bb902); branch(bb903);
branch(bb904); branch(bb905); branch(bb906);
branch(bb907); branch(bb908); branch(bb909);
money_center(mcb1000);
branch(bb1001); branch(bb1002); branch(bb1003);
branch(bb1004); branch(bb1005); branch(bb1006);
branch(bb1007); branch(bb1008); branch(bb1009);

east_coast(frb1); south_east(frb2); south_central(frb3);
east_coast(mcb100);
east_coast(bb101); east_coast(bb102); north_east(bb103);
south_east(bb104); north_central(bb105); south_central(bb106);
north_west(bb107); south_west(bb108); west_coast(bb109);
east_coast(mcb200);
east_coast(bb201); east_coast(bb202); north_east(bb203);
south_east(bb204); north_central(bb205); south_central(bb206);
north_west(bb207); south_west(bb208); west_coast(bb209);
north_east(mcb300);
east_coast(bb301); north_east(bb302); north_east(bb303);
south_east(bb304); north_central(bb305); south_central(bb306);
north_west(bb307); south_west(bb308); west_coast(bb309);
south_east(mcb400);
east_coast(bb401); north_east(bb402); south_east(bb403);
south_east(bb404); north_central(bb405); south_central(bb406);
north_west(bb407); south_west(bb408); west_coast(bb409);
north_central(mcb500);
east_coast(bb501); north_east(bb502); south_east(bb503);
north_central(bb504); north_central(bb505); south_central(bb506);
north_west(bb507); south_west(bb508); west_coast(bb509);
south_central(mcb600);
east_coast(bb601); north_east(bb602); south_east(bb603);
north_central(bb604); south_central(bb605); south_central(bb606);
north_west(bb607); south_west(bb608); west_coast(bb609);
south_central(mcb700);
east_coast(bb701); north_east(bb702); south_east(bb703);
north_central(bb704); south_central(bb705); south_central(bb706);
north_west(bb707); south_west(bb708); west_coast(bb709);
north_west(mcb800);
east_coast(bb801); north_east(bb802); south_east(bb803);
north_central(bb804); south_central(bb805); north_west(bb806);
```

```
north_west(bb807); south_west(bb808); west_coast(bb809);
south_west(mcb900);
east_coast(bb901); north_east(bb902); south_east(bb903);
north_central(bb904); south_central(bb905); north_west(bb906);
south_west(bb907); south_west(bb908); west_coast(bb909);
west_coast(mcb1000);
east_coast(bb1001); north_east(bb1002); south_east(bb1003);
north_central(bb1004); south_central(bb1005); north_west(bb1006);
south_west(bb1007); west_coast(bb1008); west_coast(bb1009);


SERVICE_PLATFORM_MAPPING_SPECIFICATION


FORALL federal_reserve  ->  route_batch_requests, route_batch_responses,
                            db_mc_balances,
                            frb_actuator_alert_on,
                            frb_actuator_alert_off,
                            frb_actuator_primary_frb_assignment,
                            frb_actuator_system_shutdown;

FORALL money_center  ->  route_requests, route_responses,
                         db_branch_balances,
                         batch_requests, send_batch_requests,
                         process_batch_requests,
                         send_batch_responses, process_batch_responses,
                         mcb_actuator_alert_on,
                         mcb_actuator_alert_off,
                         mcb_actuator_new_primary_frb,
                         mcb_actuator_system_shutdown;

FORALL branch  ->  accept_requests, send_requests_up,
                   db_account_balances,
                   receive_requests, process_requests,
                   send_responses_up,
                   process_responses, send_responses_down,
                   bb_actuator_system_shutdown;


SYSTEM_INTERFACE_SPECIFICATION


SET FederalReserveBanks
SET MoneyCenterBanks
SET BranchBanks
SET PrimaryFederalReserve
SET FederalReserveBackups
SET EastCoastBanks
SET NorthEastBanks
SET SouthEastBanks
SET NorthCentralBanks
SET SouthCentralBanks
SET NorthWestBanks
SET SouthWestBanks
SET WestCoastBanks
SET CitibankBanks
SET ChaseManhattanBanks
```

```
FederalReserveBanks = { frb1, frb2, frb3 }
MoneyCenterBanks = { i : NODE  |  money_center(i) }
BranchBanks = { i : NODE  |  branch(i) }
PrimaryFederalReserve = { frb1 }
FederalReserveBackups = { frb2, frb3 }
EastCoastBanks = { i : NODE  |  east_coast(i) }
NorthEastBanks = { i : NODE  |  north_east(i) }
SouthEastBanks = { i : NODE  |  south_east(i) }
NorthCentralBanks = { i : NODE  |  north_central(i) }
SouthCentralBanks = { i : NODE  |  south_central(i) }
NorthWestBanks = { i : NODE  |  north_west(i) }
SouthWestBanks = { i : NODE  |  south_west(i) }
WestCoastBanks = { i : NODE  |  west_coast(i) }
CitibankBanks = {mcb100, bb101, bb102, bb103, bb104, bb105, bb106, bb107,
     bb108, bb109}
ChaseManhattanBanks = {mcb200, bb201, bb202, bb203, bb204, bb205, bb206,
     bb207, bb208, bb209}


ERROR_DETECTION_SPECIFICATION


ERROR PrimaryFrbFailure
ERROR McbSecurityAttack
ERROR CoordinatedAttack
ERROR WidespreadPowerFailure

PrimaryFrbFailure =
    (EXISTS i IN PrimaryFederalReserve | node_failure(i) OR power_failure(i));

McbSecurityAttack =
    (EXISTS i IN MoneyCenterBanks  |  security_attack(i));

CoordinatedAttack  =
    ( (EXISTS i IN FederalReserveBanks  |  security_attack(i))
     AND
      (EXISTS i IN MoneyCenterBanks  |  security_attack(i)))
  OR
    ( FORALL i IN MoneyCenterBanks  |  security_attack(i));

WidespreadPowerFailure  =
    ( FORALL i IN EastCoastBanks  |  power_failure(i))
  OR
    ( FORALL i IN NorthEastBanks  |  power_failure(i))
  OR
    ( FORALL i IN SouthEastBanks  |  power_failure(i))
  OR
    ( FORALL i IN NorthCentralBanks  |  power_failure(i))
  OR
    ( FORALL i IN SouthCentralBanks  |  power_failure(i))
  OR
    ( FORALL i IN NorthWestBanks  |  power_failure(i))
  OR
    ( FORALL i IN SouthWestBanks  |  power_failure(i))
  OR
    ( FORALL i IN WestCoastBanks  |  power_failure(i));
```

```
ERROR_RECOVERY_SPECIFICATION


PrimaryFrbFailure(NODE): action_1
      PrimaryFrbFailure(NODE): action_1_1
            PrimaryFrbFailure(NODE): action_1_1_1
            McbSecurityAttack(NODE): action_1_1_2
            CoordinatedAttack(): action_1_1_3
            WidespreadPowerFailure(SET): action_1_1_4
      McbSecurityAttack(NODE): action_1_2
            PrimaryFrbFailure(NODE): action_1_2_1
            McbSecurityAttack(NODE): action_1_2_2
            CoordinatedAttack(): action_1_2_3
            WidespreadPowerFailure(SET): action_1_2_4
      CoordinatedAttack(): action_1_3
      WidespreadPowerFailure(SET): action_1_4
            PrimaryFrbFailure(NODE): action_1_4_1
            McbSecurityAttack(NODE): action_1_4_2
            CoordinatedAttack(): action_1_4_3
            WidespreadPowerFailure(SET): action_1_4_4

McbSecurityAttack(NODE): action_2
      PrimaryFrbFailure(NODE): action_2_1
            PrimaryFrbFailure(NODE): action_2_1_1
            McbSecurityAttack(NODE): action_2_1_2
            CoordinatedAttack(): action_2_1_3
            WidespreadPowerFailure(SET): action_2_1_4
      McbSecurityAttack(NODE): action_2_2
            PrimaryFrbFailure(NODE): action_2_2_1
            McbSecurityAttack(NODE): action_2_2_2
            CoordinatedAttack(): action_2_2_3
            WidespreadPowerFailure(SET): action_2_2_4
      CoordinatedAttack(): action_2_3
      WidespreadPowerFailure(SET): action_2_4
            PrimaryFrbFailure(NODE): action_2_4_1
            McbSecurityAttack(NODE): action_2_4_2
            CoordinatedAttack(): action_2_4_3
            WidespreadPowerFailure(SET): action_2_4_4

CoordinatedAttack(): action_3

WidespreadPowerFailure(SET): action_4
      PrimaryFrbFailure(NODE): action_4_1
            PrimaryFrbFailure(NODE): action_4_1_1
            McbSecurityAttack(NODE): action_4_1_2
            CoordinatedAttack(): action_4_1_3
            WidespreadPowerFailure(SET): action_4_1_4
      McbSecurityAttack(NODE): action_4_2
            PrimaryFrbFailure(NODE): action_4_2_1
            McbSecurityAttack(NODE): action_4_2_2
            CoordinatedAttack(): action_4_2_3
            WidespreadPowerFailure(SET): action_4_2_4
      CoordinatedAttack(): action_4_3
      WidespreadPowerFailure(SET): action_4_4
            PrimaryFrbFailure(NODE): action_4_4_1
```

```
            McbSecurityAttack(NODE): action_4_4_2
            CoordinatedAttack(): action_4_4_3
            WidespreadPowerFailure(SET): action_4_4_4


action_1(NODE frb_num):
action_1_1(NODE frb_num):
action_1_2_1(NODE frb_num):
action_1_4_1(NODE frb_num):
action_2_1(NODE frb_num):
action_2_1_1(NODE frb_num):
action_2_2_1(NODE frb_num):
action_2_4_1(NODE frb_num):
action_4_1(NODE frb_num):
action_4_1_1(NODE frb_num):
action_4_2_1(NODE frb_num):
action_4_4_1(NODE frb_num):
      frb_num -> shutdown();
      REMOVE(FederalReserveBanks, frb_num);
      REMOVE(PrimaryFederalReserve, frb_num);
      FederalReserveBanks -> reconfig_frb_down(frb_num);

action_1_1_2(NODE mcb_num):
action_1_2(NODE mcb_num):
action_1_2_2(NODE mcb_num):
action_1_4_2(NODE mcb_num):
action_2(NODE mcb_num):
action_2_1_2(NODE mcb_num):
action_2_2(NODE mcb_num):
action_2_4_2(NODE mcb_num):
action_4_1_2(NODE mcb_num):
action_4_2(NODE mcb_num):
action_4_2_2(NODE mcb_num):
action_4_4_2(NODE mcb_num):
      FederalReserveBanks -> raise_alert();
      mcb_num -> reconfig_mcb_attacked(mcb_num);

action_1_1_1(NODE frb_num):
action_1_1_3():
action_1_2_3():
action_1_3():
action_1_4_3():
action_2_1_3():
action_2_2_2(NODE mcb_num):
action_2_2_3():
action_2_3():
action_2_4_3():
action_3():
action_4_1_3():
action_4_2_3():
action_4_3():
action_4_4_3():
      BranchBanks -> shutdown();
      MoneyCenterBanks -> shutdown();
      FederalReserveBanks -> shutdown();

action_1_1_4(SET region):
```

```
action_1_2_4(SET region):
action_1_4(SET region):
action_1_4_4(SET region):
action_2_1_4(SET region):
action_2_2_4(SET region):
action_2_4(SET region):
action_2_4_4(SET region):
action_4(SET region):
action_4_1_4(SET region):
action_4_2_4(SET region):
action_4_4(SET region):
action_4_4_4(SET region):
      // if primary frb in region, promote another FRB
      // if any mcbs in region, promote BBs in other region
      switch(region)
            case east_coast:
                  frb1 -> shutdown();
                  REMOVE(FederalReserveBanks, frb1);
                  FederalReserveBanks -> reconfig_frb_down(frb1);
                  mcb100 -> shutdown();
                  REMOVE(MoneyCenterBanks, mcb100);
                  bb103 -> promote_to_mcb();
                  ADD(MoneyCenterBanks, bb103);
                  CitibankBanks -> reconfig_mcb_down(mcb100, bb103);
                  mcb200 -> shutdown();
                  REMOVE(MoneyCenterBanks, mcb200);
                  bb203 -> promote_to_mcb();
                  ADD(MoneyCenterBanks, bb203);
                  ChaseManhattanBanks -> reconfig_mcb_down(mcb200, bb203);
```


PDF