# Error Recovery in Critical Infrastructure Systems

John C. Knight, Matthew C. Elder, Xing Du
*Department of Computer Science*
*University of Virginia*
*Charlottesville, VA*
*{knight, elder, xd2a}@cs.virginia.edu*

***Abstract***

*Critical infrastructure applications provide services upon which society depends heavily; such applications require survivability in the face of faults that might cause a loss of service. 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. Much of the literature on fault-tolerant distributed systems focuses on local error recovery by masking the effects of faults. We describe a direction for error recovery in the face of catastrophic faults where the effects of the faults cannot be masked using available resources. The goal is to provide continued service that is either an alternate or degraded service by reconfiguring the system rather than masking faults. We outline the requirements for a reconfigurable system architecture and present an error recovery system that enables systematic structuring of error recovery specifications and implementations.*

## 1. Introduction

The provision of dependable service in infrastructure applications such as electric power generation and control, banking and financial systems, telecommunications, and transportation systems has become a major national concern [34], [35]. Society has become so dependent on such services that the loss of any of them would have serious consequences. Such services are often referred to as *critical infrastructure applications*.

As has occurred in many domains, sophisticated information systems have been introduced into critical infrastructure applications as the cost of all forms of computing hardware has dropped and the availability of sophisticated software has increased. This has led to dramatic efficiency improvements and service enhancements but, along with these benefits, a significant vulnerability has been introduced: the provision of service is now completely dependent in many cases on the correct operation of computerized information systems. Failure of an information system upon which a critical infrastructure application depends will often eliminate service quickly and completely. The dependability of the information systems, which we refer to as *critical information systems*, has therefore become a major concern.

Dependability has many facets—reliability, availability and safety, and so on [24]—and critical infrastructure applications have a variety of dependability requirements. In most cases, very high availability is important, but reliability and safety arise in such systems as transportation control, and security is becoming an increasingly significant dependability property in all application domains.

An important dependability requirement of critical infrastructure applications is that, under predefined adverse circumstances that preclude the provision of entirely normal service to the user, such systems must provide predefined forms of *alternate* service. The necessary service might be a degraded form of normal service, a different service, or some combination. Adverse circumstances might be widespread environmental damage, equipment failures, software failures, sophisticated malicious attacks, and so on. This particular dependability requirement is referred to as *survivability.* In the terminology of fault tolerance, survivability can be thought of as requiring very specific (and usually elaborate) error recovery after a fault.

This paper is about the mechanisms that are needed within the applications themselves to provide error recovery, i.e., state restoration and continued service, under circumstances where the application has been subjected to extreme damage. Application systems must be designed to permit effective state restoration and appropriate continued service, and we address the issues of application system design in this context. We are concerned with faults that affect either large parts of the system or the entire system and which cannot be masked. Thus, faults such as the failure of a single processor or a single communications link are not within the scope of this paper. Faults such as these can be tolerated by local redundancy and their effects can be totally masked at reasonable cost. Faults of interest include widespread physical damage in which substantial resources are lost, and coordinated security attacks in which multiple attacks occur in a short time period. We assume that error detection and damage assessment are taken care of by some other mechanism such as a control-system architecture [45].

The outline of the paper is as follows. In the next section we summarize briefly the functionality and characteristics of critical infrastructure applications and their associated information systems, and then present a detailed example of survivability. In section 4, we discuss the role of fault tolerance and the requirements for an approach to survivability. In section 5, we present some related work, and in section 6 we define directions for a solution approach. Finally, we present our conclusions.

## 2. Critical infrastructure applications

Some background material about critical infrastructure applications is helpful in understanding the technical approaches that might be developed to realize survivability. Detailed descriptions of four applications are available elsewhere [20]. In this section we summarize three applications very briefly and then outline a set of important characteristics that tend to be present in information systems supporting critical infrastructure applications. Finally in this section, we discuss characteristics of future critical infrastructure applications that impact approaches to survivability that might be developed.

### 2.1. Applications

The nation’s banking and finance systems provide a very wide range of services—check clearing, ATM service, credit and debit card processing, securities and commodities markets, electronic funds transfers, foreign currency transfers, and so on. These services are implemented by complex, interconnected, networked information systems.

The most fundamental financial service is the payment system. The payment system is the mechanism by which value is transferred from one account to another. Transfers might be for relatively small amounts, as occur with personal checks, all the way up to very large amounts, typical of commercial transactions. For a variety of practical and legal reasons, individual banks do not communicate directly with each other to transfer funds. Rather, most funds are transferred in large blocks by either the Federal Reserve or by an Automated Clearing House (ACH).

The freight-rail transport system, another critical infrastructure application, moves large amounts of raw materials, manufactured good, fuels, and food. Although not responsible for moving everything in any one of these categories, loss of freight-rail transportation would be devastating. Management of the freight-rail system uses computers extensively for a variety of purposes. For example, every freight car in North America is tracked electronically as it moves and databases are maintained of car and locomotive locations. Tracking is achieved using track-side equipment that communicates in real time with computers maintaining the database.

An especially important application that is being used increasingly in the freight-rail system is just-in-time delivery. Train movements are scheduled so that, for example, raw materials arrive at a manufacturing plant just as they are required. This type of service necessitates analysis of demands and resources over a wide area, often nationally, if optimal choices are to be made.

The generation and distribution of electric power, the third critical infrastructure application we consider, is accomplished by a wide variety of generating, switching, and transmission equipment that is owned and operated by a number of different utility companies. However, all of this equipment is interconnected, and control is exercised over the equipment using a system that is rapidly becoming a single national network. The control mechanisms within a region are responsible for managing the equipment in that region and interconnection of area control mechanisms is responsible for arranging and managing power transfers between regions. The complexity of the control mechanisms in the power generation industry is being affected considerably by industry deregulation.

## 2.2. Application system characteristics

The architecture of the information systems upon which critical infrastructure applications rely are tailored very substantially to the services of the industries which they serve and influenced inevitably by cost-benefit trade-off's. For example, the systems are typically distributed over a very wide area with large numbers of nodes located at sites dictated by the application. Beyond this, however, there are a number of characteristics that these applications possess in whole or in part which are important in constraining the ways by which these applications approach error recovery. These characteristics are as follows:

- *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.

- *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.

- *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.

- *Performance requirements.* Some critical infrastructures applications, such as the financial payment system, have soft real-time constraints and throughput requirements (checks have to be cleared and there are lots of checks) 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.

- *Extensive databases.* Infrastructure applications are all about data. Many employ several very 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.

## 2.3. Future system characteristics

The characteristics listed in the previous section 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 also. We hypothesize that the following will be important architectural aspects of future infrastructure systems:

- *Very large number 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 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. Survivability—an example

In this section, we present an example of survivability requirements to illustrate the extent, scope, and complexity of the error recovery that might well be needed in a typical critical infrastructure application. We use as an example application a *highly* simplified version of the national financial payment system. It is important to note that, inevitably, most of the details of the payment system are missing from this example and simplifications have been made since we seek only to illustrate certain points. The interested reader is referred to the text by Summers for comprehensive details of the payment system [46]. In addition, the faults and continued service requirements in this example are *entirely* hypothetical and designed for illustration only, but characteristic of strategies that might be employed for error recovery.

### 3.1. System architecture and application functionality

The information system that implements a major part of the payment system is very roughly a hierarchic, tree-like network as illustrated in Figure 1. At the top level is a central facility operated by the Federal Reserve. At the second level are the twelve regional Federal Reserve banks. At the third level of the tree are the approximately 9,500 commercial banks that are members of the Federal Reserve. Finally, at the lowest level of hierarchy are the remaining 16,500 or so banks and their branch banks [20].

Processing a retail payment (an individual check) in this system proceeds roughly as follows. At the lowest level, nodes simply accept checks, create an electronic description of the relevant information, and forward the details to the next level in the hierarchy. At the next level, payments to different banks are collected together in a batch and the details forwarded to the Federal Reserve system. Periodically (typically once a day) the Federal Reserve moves funds between accounts that it maintains for commercial banks and then funds are disbursed

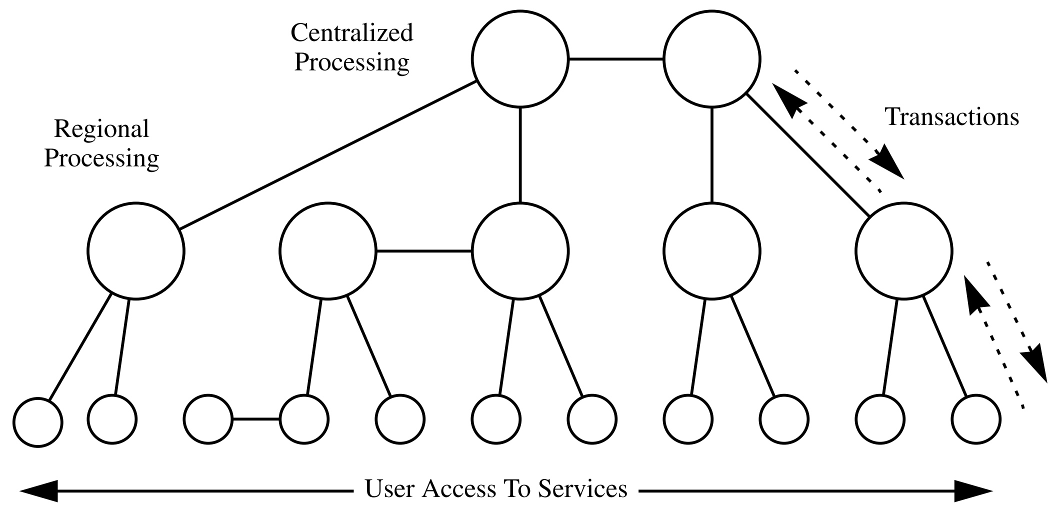

**Figure 1. Example application network topology**

down through the system to individual user accounts. Large commercial payments originate electronically and are handled individually as they are presented.

Extensive amounts of data are maintained throughout this system. User account information is maintained at central facilities by retail banks, and this information provides all the expected user services together with check authentication (as occurs when a check is scanned at a retail outlet). The Federal Reserve maintains accounts for all its member banks together with detailed logs and status information about payment activity. But, of course, this is just a small part of the data maintained by the banking and financial system. Vast amounts of data are also needed for all the other financial services, and the databases are used in combination and in different ways by different services.

## 3.2. Survivability requirements

A complete survivability specification must document precisely all of the faults that the system is required to handle and, for each, document the prescribed system response. Hypothetical examples of the possible faults and their high-level responses for our simplified version of the payments system are shown in Table 1. We include in the table faults ranging from the loss of a single leaf node to the loss of the a critical node and its backup facilities.

For purposes of illustration, we examine one particular fault in more detail to see what error recovery actions are needed. The fault we use for illustration is the loss of the top-level node of the financial payment system—the Federal Reserve system's main data center and its backup facilities. Using our highly simplified architecture of the payment system in this example, we assume that this node consists of a single processing entity with a single backup that maintains mirror image databases. The actual Federal Reserve system uses a much more sophisticated backup system. The survivability requirements for this fault are the following:

- *Fault:* Federal Reserve main processing center failure (common-mode software failure, propagation of corrupt data, terrorism).

- *On failure:* Complete suspension of all payment services. Entire financial network informed (member banks, other financial organizations, foreign banks, government agencies). Previously identified Federal Reserve regional bank designated as temporary replacement for Federal facilities. All services terminated at replacement facility, minimal payment service started at replacement facility (e.g., payment service for federal agencies only). All communication redirected for major client nodes.

- *On repair:* Payment system restarted by resuming applications in sequence and resuming service to member banks in sequence within an application. Minimal service on replacement facility terminated.

For this particular fault, we assume that all processing ceases immediately. This is actually the most benign fault that the system could experience at the top-level node. More serious faults that could occur include undetected hardware failure in which data was lost, a software fault that corrupted primary and backup databases, and an operational failure in which primary data was lost.

State restoration in this case involves establishing a consistent state at all of the clients connected to the Federal Reserve system. This requires determining the transaction requests that have been sent to the Federal Reserve but not processed. Since this is a standard database

**Table 1. Survivability requirements summary**

| Fault | Response |
|---|---|
| Single local bank fails (local power failure, hardware failure, operator error). | *On failure:* Local bank ceases service. Regional center buffers transactions for local bank.<br>*On repair:* Local bank informs regional center. Regional center transmits transaction backlog. Regional center resumes transaction transmission. Local branch resumes normal service. |
| Multiple local banks fail (wide-area power failure, wide-area environmental stress, common-mode software failure). | *On failure:* Local banks cease service. Regional center buffers transactions for local banks. Regional center starts minimal user services (e.g., electronic funds transfer for selected major customers only).<br>*On repair:* Local banks inform regional center as they are repaired. Regional center transmits transaction backlog. Regional center resumes transaction transmission. Local branch resumes normal service. Regional center terminates minimal user services. |
| Security penetrations of multiple local banks associated with a single commercial bank (coordinated security attack). | *On failure:* Each local bank ceases service and disconnects itself from the network when its local intrusion alarm is raised. Federal Reserve suspends operations with commercial member bank under attack and buffers all transactions for that bank.<br>*On repair:* Reset all nodes owned by commercial bank under attack. All nodes change cryptographic keys and switch to aggressive intrusion detection. System-wide restart of crucial services only. |
| Regional center primary site fails (power failure, hardware failure, software failure, operational error, environmental stress). | *On failure:* Primary site ceases service. Backup site starts service. All connected nodes informed. All communications—up to Federal system and down to branches—switched from primary to backup. Appropriate service selection made—full or reduced. Appropriate switch selection made—instantaneous or delayed.<br>*On repair:* Primary site informs all connected nodes that it is repaired. Primary site databases synchronized. Communications switched. Primary site resumes services in prescribed priority order. Backup site ceases service. |
| Regional center primary *and* backup sites fail (wide-area power failure, common-mode software failure, wide-area environmental stress, terrorism). | *On failure:* Primary and backup sites cease service. All connected nodes informed. Previously identified local bank processing center designated as temporary replacement for regional facilities. All services terminated at replacement facility, minimal regional service started at replacement facility (e.g., account services for commercial and government clients only).<br>*On repair:* Regional service restarted by resuming applications in sequence and resuming service to local banks in sequence within an application. Minimal service on replacement facility terminated. |

issue, we do not consider it further. We note, however, that notification or detection of failure by the clients is an essential element of error recovery.

The provision of continued service is more complex in this fault scenario. Given the loss of the most critical node in the network with a tree topology, the network is now effectively partitioned. One part of error recovery will involve re-establishing connectivity between the partitioned subtrees, each of which has a Federal Reserve regional bank at its top node. There are a number of alternatives for re-establishing connectivity:

- Promote one regional bank to be the new root node in the tree and have all other regional banks establish links to it. (This requires 11 new links to be established.)
- Establish links between each pair of regional banks, resulting in a fully-interconnected 12-node network. (This requires 66 new links to be established.)
- Establish links to provide some other topology; for example, connect the 12 regional banks with a ring topology. (This requires 12 new links to be established.)

In practice, it is unlikely that any of the above three alternatives would be used. A combination using different strategies in different locations is the most likely approach and it is very possible that even in this case parts of the network would remain unconnected.

Once whatever connectivity that is possible is reestablished, a major reorganization through the network would be required. First, the new root of the tree will have to suspend most but probably all of its normal service activities. Second this node will have to prepare the copies of the databases that are needed for payment processing and that it would have to have maintained during normal processing if it were to function as an emergency backup. Third, the entire set of clients will have to be informed of the change and of the level of service that will be provided and when this will occur. These clients will have to take their own actions including eliminating many services, reducing others, and perhaps starting certain emergency services. Fourth, the new root node will have to initiate payment applications up to the limit of the processing and communications capacity that was available. The available capacity will be greatly reduced and the services that the system could provide once operational would be far less that would normally be the case. Which customers get what service would have to be have been defined ahead of time as part of the survivability specification.

Much more complex but probably more useful recovery scenarios are possible for critical services if a restricted form of service is acceptable. For example, one of the services of the Federal Reserve is maintenance of member bank accounts. Maintenance of member bank accounts could be taken over by the temporary replacement node but, because of reduced facilities, services would be severely reduced. This function could be distributed following a catastrophic failure, however. We consider two possible solution strategies:

- Each member bank maintains only its own account details, and only sends a batch message to be processed if it has the funds to cover the deposit. (This approach distributes responsibility for maintaining positive balances.)
- Each regional bank maintains an account balance for every other regional bank with which it exchanges batch messages. (This approach requires more resources, but allows more value to be transferred throughout the system.)

The details of the fault scenario we have described in this example are possible from the computer science perspective as are many others. What the banking community requires in practice depends upon the many details and priorities that exist within that domain and will probably be far more elaborate that our example. However, our example does illustrate many of the issues that have to be considered in application error recovery and shows how complex this process is.

An important aspect of survivability that is omitted form this example is the need to cope with multiple sequential faults. It will be the case in many circumstances that a situation gets worse over time. For example, a terrorist attack on the physical equipment of a critical information system might proceed in a series of stages. The attack might be detected initially during an early stage (although it is highly unlikely that the detection process would be able to diagnose a cause in this case) and the system would then take appropriate action. Subsequent failures of physical equipment would have to be dealt with by a system that had already been reconfigured to deal with the initial attack. This complicates the provision of error recovery immensely.

## 4. Survivability and fault tolerance

### 4.1. The role of fault tolerance

In general, survivability is a requirement or a set of related requirements that a system must meet. As illustrated by the example above, the requirements can be quite complex and will often involve many different aspects of the application. For different faults that a system might experience, the requirements that have to be met might be quite different from each other and require entirely different actions by the application. In particular, for critical infrastructure applications, it will often be the case that the effects of a fault leave the system with greatly reduced resources (processing services, communications capacity, etc.) and substantial changes in the service provided to the user will be necessary.

Fault tolerance is one of the mechanisms by which dependability (and thus survivability) can be obtained. It is not the only mechanism of course since fault elimination and fault removal are often alternate (and complementary) possibilities. In the case of critical infrastructure applications, however, the manifestation of faults during operation is inevitable. Such systems cannot be protected against all environmental damage, terrorist acts, operational mistakes, software defects, and so on. Thus, adding the ability to tolerate certain types of fault is the only practical approach to achieving survivability.

Tolerating a fault does not necessarily mean masking its effects, however. The essential meaning of survivability is the ability of a system to deal with more “serious” faults by providing a prescribed service that is not the same as the normal service. The catastrophic faults to which a system of interest must respond are those that are not masked, i.e. by design there is insufficient redundancy for the system to be able to handle the faults transparently. If it were intended that the system mask such faults, as will be the case for many faults, then the system would do so. Catastrophic faults that are not masked in any given system are not necessarily unanticipated. Rather, a conscious decision is made in favor of an approach to error recovery other than masking because the cost of handling faults transparently is redundancy. And redundancy is expensive. Some redundancy is necessary even if a fault is not to be masked, and, in addition, redundancy is necessary for the detection of errors. However, replicating all the elements of a critical information system so that all faults of interest can be masked is prohibitively expensive.

Since we are concerned with faults whose effects will not be masked, fault tolerance in general requires actions by the application. The particular actions required in any given system are application-specific but they will require such functionality as stopping some services, starting others, and modifying yet others. In order to make such changes, the application must be prepared to make the changes and so must be designed with this in mind.

For purposes of analysis, we view an application executing on a distributed system as a concurrent program with one or more processes running on each node and with processes communicating via a protocol that operates over network links. Although there is no shared memory in the conventional sense, it is very likely that there are shared files. This view is useful because it permits existing work on fault tolerance in concurrent systems to act as a starting point from which to develop application error recovery mechanisms in the sense that we desire.

## 4.2. Requirements for a solution

The general requirements that have to be met by any realistic approach to application error recovery derive from the characteristics of the applications and the need to tolerate serious faults that cannot be masked. More precisely, we identify the following solution requirements:

- *Very large networks running sophisticated applications must be supported.* The scale of critical infrastructure applications necessitates a solution approach that scales to networks with thousands of nodes. Similarly, the size of the application software presents significant performance challenges that must be met.

- *Resuming normal processing following the repair of fault must be supported.* The problem of dealing with the effects of a fault is really only half of the problem. The systems of interest typically have very high availability requirements and will be repaired while in operation. Thus, resuming prescribed levels of service after repair must be part of any viable solution.

- *There must be minimal application re-write required.* For many reasons—economic, political, technical, and otherwise—it is nearly impossible to re-write the software within critical information systems from scratch. Critical information systems will most likely have to evolve to provide improved survivability; however, the extensive use of COTS and legacy software currently in these systems complicates revision of the software. Revision to the applications must be kept to a minimum, but the application will have to make provision for support of new error recovery services.

- *Error recovery must be performed securely.* Security attacks against these systems are a major threat. It must be the case that any new error recovery services that are provided do not introduce an additional security vulnerability that can be exploited to perpetrate further damage to the system.

- *Link failure and partitioning are errors that must be handled.* The stylized connection structure of most critical infrastructure applications dictates that, by default, every node will not be able to communicate with every other node. This introduces the possibility that a link failure will partition the network. Because the service provided is composed of functionality provided by multiple application nodes, error recovery must include strategies for circumventing link failure and network partitions.

- *There are highly structured requirements for continued service.* The requirements for continued service will not be homogenous—in fact they will be far from it. Different nodes and different node classes will be required to implement different services. Similarly, nodes in different geographic regions might be called upon to act differently after a fault. Complicating the problem even further is the virtual certainty that different services on different nodes will have to be coordinated and that the coordination might have a hierarchic structure given the topology of the system.

In seeking an approach to meeting these requirements, we begin by reviewing previous work in a variety of areas and then we proceed to discuss a direction for solution.

## 5. Related work

Developments in several technical fields can be exploited to help deal with the problem of error recovery in distributed applications. In this section we review related work in the areas of system-level approaches to fault tolerance, fault tolerance in wide-area networking applications, reconfigurable distributed systems, and system specification and architectural description languages.

### 5.1. System-level approaches to fault tolerance

Jalote presents an excellent framework for fault tolerance in distributed systems [19]. Jalote structures the various services and approaches to fault tolerance into levels of abstraction. The layers, from highest to lowest, of a fault-tolerant distributed system according to Jalote are shown in Figure 2. Each level of abstraction provides services for tolerating faults, and in most cases there are many mechanisms and approaches for implementing the given abstraction. At the lowest level of abstraction above the distributed system itself are the building blocks of fault tolerance, including fail-stop processors, stable storage, reliable message delivery, and synchronized clocks. One level above that is another important building block—reliable and atomic broadcast; different protocols provide different guarantees with respect to reliability, ordering, and causality of broadcast communication. The levels above that provide the services upon which systems can be built to tolerate certain types of fault, including abstractions for atomic actions and processes and data resilient to low-level failures. Finally, the highest level of abstraction enables tolerance of design faults in the software itself.

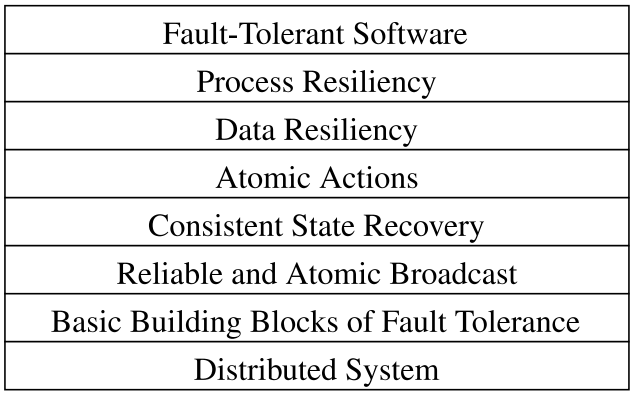
Fault-Tolerant Software
Process Resiliency
Data Resiliency
Atomic Actions
Consistent State Recovery
Reliable and Atomic Broadcast
Basic Building Blocks of Fault Tolerance
Distributed System

**Figure 2. Levels of a fault-tolerant distributed system [19]**

Given this framework for fault tolerance in distributed 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.

**5.1.1. Cristian/Advanced Automation System:** Cristian provided a survey of the issues involved in providing fault-tolerant distributed systems [11]. 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 [14]. Cristian described the primary requirement of the air traffic control system as ultra-high availability and stated that the approach taken is 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 [14].

**5.1.2. 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 contains 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 [8].

Work on ISIS proceeded in subsequent years resulting in another group communications system, Horus. The primary benefit of Horus over ISIS is a flexible communications architecture that can be varied at runtime to match the changing requirements of the application and environment. Horus achieves this flexibility using a layered protocol architecture in which each module is responsible for a particular service [47]. Horus also works with a system called Electra, which provided a CORBA-compliant interface to the process group abstraction in Horus [26]. Another system that built on top of Electra and Horus together, Piranha, provided high availability by supporting application monitoring and management facilities [27].

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 [48].

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 [38], [39], [40].

**5.1.3. Other system-level approaches:** Another example of fault tolerance that focuses on communication abstractions is the work of Schlichting. 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 [7].

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 [9].

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 [42]. The atomic actions ensure that all operations support the properties of serializability, failure atomicity, and permanence of effect.

## 5.2. Fault tolerance in wide-area network applications

Fault tolerance is typically applied to relatively small-scale systems, dealing with single processor failures and very 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 [3].

The Eternal system, built 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 [32].

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 [6].

## 5.3. 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 addressed. However, the most serious fault—the catastrophic, non-maskable fault—is not addressed in any of the previous 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 explored in this section.

**5.3.1. 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 [22].

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 remains on the incremental changes to a distributed system configuration for evolutionary purposes [23].

The successor to CONIC, Darwin, is a configuration language that separates program structure from algorithmic behavior [31]. 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 [28].

**5.3.2. 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 [36].

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 [18].

Welch and Purtilo have extended Hofmeister's work in a particular application domain, Distributed Virtual Environments. They utilize 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 [49].

### 5.4. System specification and architectural description languages

Finally, many issues related to system and software architecture arise in the description and analysis of complex distributed systems. It is helpful, therefore, to be able to describe these architectures in a concise and comprehensive manner. A specification technology that suggests itself for systems of such magnitude is architectural description languages (ADLs). ADLs describe a particular piece of a system, the architecture in terms of components and connectors. A focus of architectural description languages is often that the specification of the architecture provide a mechanism to facilitate automatic construction of the distributed system [16].

Magee and Kramer's work on Darwin is an example of an architectural description language intended to facilitate construction of distributed systems. The Darwin language has a precise semantics that enables the construction of distributed systems from their specification in Darwin. The work has evolved from specifying and executing dynamic changes in a distributed system to specification of a distributed system and generation of its implementation; though Darwin still provides facilities for specification of dynamic change, elaboration of a Darwin specification is an important goal as well [30].

There are many other architectural description languages, each with its own focus and goals. Allen and Garland proposed the Wright architectural specification language in order to support direct specification and analysis of architectural description. A key idea behind Wright is that interaction relationships between components of a software system should be directly specified, in the case of Wright as protocols describing the interaction [2].

Shaw et al. proposed an informal model for an architectural description language, then developed an initial system for architectural description, UniCon. Their work is based upon recognizing common patterns in software architectures, such as pipe and filter, implicit invocation, etc., and then providing abstraction mechanisms for those element types [41].

Luckham et al. developed a specification system for prototyping architectures of distributed systems using an event-based, concurrent, object-oriented language, Rapide. Rapide allows the simulation and behavioral analysis of software system architectures during the development process [25].

Finally, Moriconi and Qian explored the process of transforming an abstract software architecture into an instance architecture correctly and incrementally using refinements [33].

## 6. A solution direction

Despite the many results that have been achieved in survivability and related fields, existing technology does not satisfy the requirements for a solution that were identified in Section 4.2. In this section, we describe a solution direction for error recovery (state restoration and continued service) in critical information systems following catastrophic, non-masked failures. The solution direction is conceptual and is based on previous work in the areas of fault tolerance and reconfigurable distributed systems. Many detailed issues are identified in the development of this solution direction.

Figure 3 shows the general layered structure that we anticipate in a single node. Note that our solution direction does not address *local* faults. By local, we mean faults that affect a single hardware or software component. We assume that all faults that are local are dealt with by some mechanism that masks their effects. In Figure 3 this functionality would be provided by mechanisms associated with the local recovery layer. Thus synchronized, replicated hardware components are assumed so that losses of single processors, disk drives, communications links, and so on are masked by hardware redundancy. If necessary, more sophisticated techniques such as virtual synchrony can be used to ensure that the application is unaffected by local failures.

The faults with which we are concerned, faults that are not masked, are dealt with by the global recovery layer shown in Figure 3. The high-level concept that we suggest is the use of formal specification to define the survivability requirements together with synthesis of the

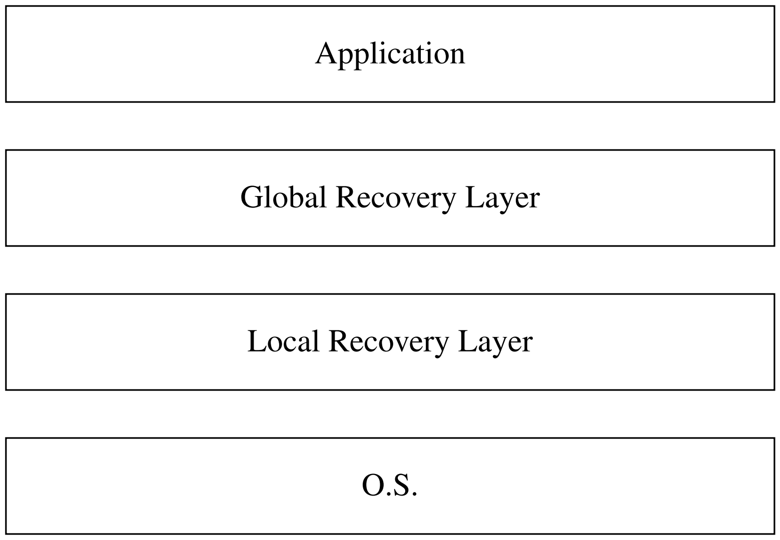

**Figure 3. Node abstraction layers**

implementation of the application. The synthesized system would use the global recovery layer to achieve error recovery.

We begin by explaining this concept and then review the details including the system software architecture used in each node. Finally, we discuss some of the implementation details.

### 6.1. Specification and synthesis

The size of current and expected critical information systems, the variety and sophistication of the services they provide, and the complexity of the survivability requirements means that a solution approach that depends upon traditional development techniques is infeasible in all but the simplest cases. The likelihood is that future systems will involve 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 such a system using conventional methods is quite impractical, and so our solution direction is based on the use of a formal specification to describe the required application reconfiguration and the use of synthesis to generate the implementation from the formal specification. The approach is illustrated in Figure 4.

There are many advantages to working with specifications rather than implementations. First and foremost is the ability to specify solutions at a high-level, thereby abstracting away to some extent the details of working with so many nodes, of so many different types, that provide so many different services. An implementation-based solution would require too much work, dealing with such a wide variety of nodes, applications, errors, and recovery strategies at a lower level. In addition, specifications provide the ability to reason about and analyze solutions at a higher level. An implementation synthesized from a specification also allows recovery strategies to be changed quickly; different error recovery schemes can be rapidly prototyped and explored using a specification-based approach.

Precise specification of the error recovery in a critical information system is a complex undertaking. It involves three major sub-specifications: (1) the topology of the system and a

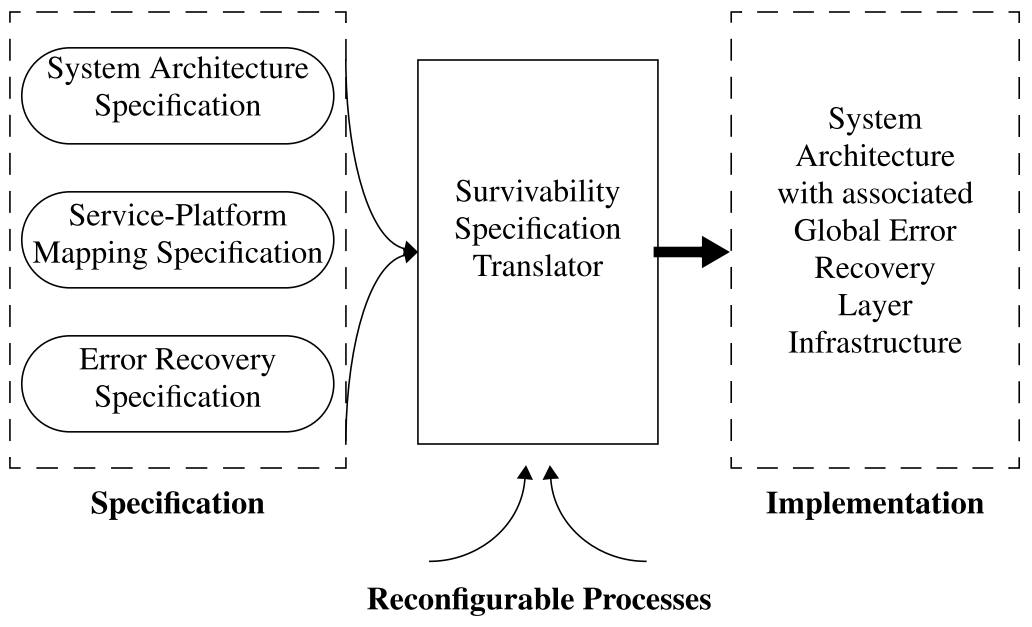
System Architecture Specification
Service-Platform Mapping Specification
Error Recovery Specification
Specification
Survivability Specification Translator
System Architecture with associated Global Error Recovery Layer Infrastructure
Implementation
Reconfigurable Processes

**Figure 4. Error recovery system**

detailed description of the architecture and platform; (2) an abstraction of the services each node supplies to the system and the mapping of services to platform for cases involving full functionality and degraded or alternate service; and (3) a specification of the necessary state changes from any acceptable system reconfiguration to any other in terms of topology, functionality, and geometry (assignment of services to nodes). More precisely, the sub-specifications that together make up the input to the translator are the following:

- *System Architecture Specification (SAS).* The system architecture specification describes the nodes and links, including 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, for example.

- *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.

- *Error Recovery Specification (ERS).* The overall structure of the specification is that of a finite-state machine that characterizes the requisite responses to each fault. Arcs are labelled with faults and show the state transitions for each fault from every relevant state. The actions associated with any given transition are extensive because each action is essentially a high-level program that implements the error recovery component of the full system survivability specification. The full system survivability specification names the different states (system environments) that the system can be in, including the errors that will be detected and handled. The ERS takes this list of system states and describes the actions—i.e., reconfigurations—that must be performed when the system transitions from one environment to another. The ERS uses the SAS and the SPMS to describe the different system configurations and alternate service modes under each system state.

We make no specific comments about the syntax that might be used for these notations. However, existing work on architectural description languages provides an excellent starting point for the development of syntax and precise semantics.

### 6.2. Reconfigurable processes

We define a specialized type of application process, the *reconfigurable process*, that is used as the building block for critical information systems. The key specialization is that a reconfigurable process supports certain *critical services* that are needed for error recovery in addition to implementing some aspect of the required system functionality. A critical information system is then a collection of reconfigurable processes that cooperate in the normal way to implement normal application functionality. However, they can be manipulated using their critical-service interfaces to prepare for error recovery and to effect that recovery.

The importance of the addition of critical services is that they are the basic services needed for reconfiguration and they are available with every process. Thus the survivability specification need not be concerned with the idiosyncrasies of individual node functionality. As an example of critical service, consider the obvious implementation requirement that some processes in a system undergoing error recovery will need to be started and others stopped. A critical service that processes must provide is the ability to be started and another is the ability to be stopped. Neither of these actions is trivial, in fact, and neither can be left to the basic services of the operating system.

A second more detailed example of a critical service arises in the provision of backward error recovery. In the event that a system designer wishes to exploit a backward-error recovery mechanism, he or she will want to be sure that all the processes involved are capable of establishing recovery points and that groups of processes are capable of discarding them in synchrony. Since this is such a basic facility in the context of error recovery, a set of critical services is required to permit the manipulation of recovery points.

The following is a preliminary list of the critical services that a reconfigurable process has to support:

- Start, suspend, resume, terminate, delay.
- Change process priority.
- Report prescribed status information.
- Establish recovery point, discard recovery point.
- Effect local forward recovery by manipulation of local state information (e.g., reset the state).
- Switch to an alternate application function as specified by a parameter.
- Database management functions such as synchronizing copies, creating copies, withdrawing transactions, and restoring a default state.

The critical services are conceptually simple in many cases but this simplicity is deceptive. Many application processes will include very extensive functionality and this functionality does not necessarily accommodate services such as process suspension. Far worse are situations that involve processes which manipulate databases. Such processes will have to be very carefully developed if the creation of a checkpoint is to be efficient.

## 6.3. Node architecture

Each node in a critical information system that supports comprehensive error recovery using the approach outlined here will have a fixed (but not a restrictive) architecture. The basic application will be constructed in a standard manner as a collection of processes each of which is enhanced to support critical services. Under benign circumstances, these processes execute in a normal manner and provide normal functionality. Their critical services will be used periodically in a proactive manner to make provision for some form of recovery such as the establishment of recovery points or the forced synchronization of a database with backup copies.

The most obvious architectural requirement that has to be met at each node is that the node architecture support the provision of the various forms of degraded service associated with each fault. The software that implements degraded service is provided by application or domain experts, and the details (functional, performance, design, etc.) of this software are not part of the approach being outlined here. In practice, the way in which the software that provides degraded service is organized is not an issue either. The various degraded modes could be implemented as cases within a single process or as separate processes, as the designer chooses.

The interface between the node and the error detection mechanism (the control system) is a communications path from the control system to an *actuator* resident on the node. The actu-

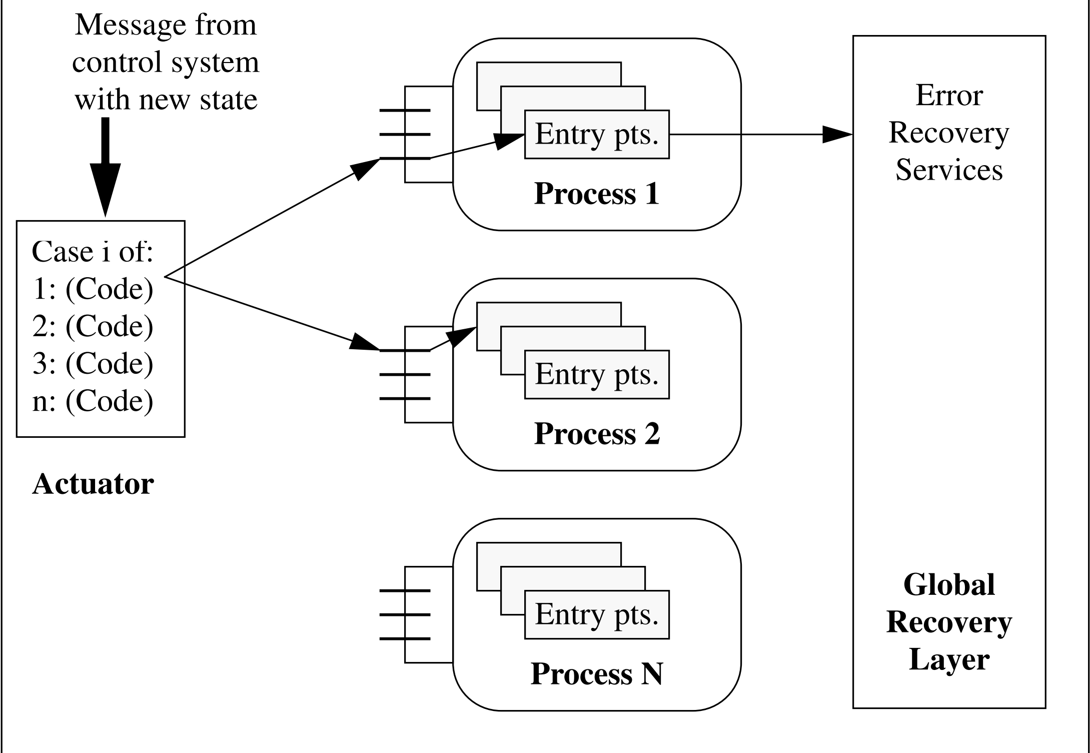

**Figure 5. Node architecture (with reconfigurable processes)**

ator is a process that accepts notifications from the control system about erroneous states and undertakes the actions needed on that node to cope with the errors. Thus, the actuator implements the changes dictated by the survivability specification, and it does this by making the necessary changes to the node's software using the critical services of the various reconfigurable processes. The actuator implementation is synthesized by the survivability specification translator.

### 6.4. Critical service implementation

The critical services provided by a reconfigurable process are implemented by the process itself in the sense that the service is accessed by a remote procedure call (or similar) and a mechanism internal to the process implements the service. The exact way in which the implementation is done will be system specific but an obvious layered architecture that supports this implementation suggests itself (see Figure 3).

The global recovery layer provides the interface that is used in the implementation of critical services within reconfigurable processes. The following is a preliminary list of the functions that the global recovery layer has to support:

- Process synchronization.
- Inter-process communication.
- Multicast to a set of processes.
- Establishment of a checkpoint for a process.
- Establishment of a set of coordinated checkpoints for a group of processes.
- Restoration of the state of a process from a checkpoint.
- Restoration of the states of a group of processes from a set of checkpoints.
- Reset of a process' state in support of forward error recovery.
- Synchronizing two or more processes to establish lock-step operation.
- Redirection of communication.

The global recovery layer would provide these services in a largely application-independent manner. Thus, a common global recovery layer implementation could be used by multiple applications with initial configuration achieved by generation parameters such as a process name table and target system topology.

## 7. Conclusion

In this paper, we have explored the problem of error recovery in critical infrastructure applications, from needs to solutions. We have described the problem context (system characteristics and survivability requirements) and solution framework (fault tolerance) in order to better understand the constraints on and requirements of a solution. Finally, we suggest a solution direction involving system specification and generation in an error recovery system.

This work is being conducted in the context of general survivability research at the University of Virginia. As mentioned previously, the error detection and damage assessment phases of fault tolerance are handled by a control-system architecture—hierarchical, adaptive, and overlayed upon the critical information system [45]. In addition, we have developed a modelling and simulation framework to enable experimentation on example critical information systems [44]. A model of the financial payments system has been constructed using this experimentation system, and experiments and evaluation on both error detection and error recovery strategies are being conducted using this system.

## 8. Acknowledgments

It is a pleasure to thank Kevin Sullivan for numerous discussions about the ideas expressed in this paper. This effort was sponsored 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 purposes 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. This work was also done under the support of a National Science Foundation Research Fellowship. Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.

## 9. References

[1] Agnew, B., C. Hofmeister, and J. Purtilo. “Planning for Change: A Reconfiguration Language for Distributed Systems,” Proceedings of the 2nd International Workshop on Configurable Distributed Systems, IEEE Computer Society Press, Los Alamitos, CA, March 1992, pp. 15-22.

[2] Allen, R. and D. Garlan. “Beyond Definition/Use: Architectural Interconnection,” ACM SIGPLAN Notices, Vol. 28 No. 9, August 1994, pp. 35-45.

[3] 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.

[4] Anderson, T. and J. Knight. “A Framework for Software Fault Tolerance in Real-Time Systems,” IEEE Transactions on Software Engineering, Vol. SE-9 No. 3, May 1983, pp. 355-364.

[5] Anderson, T. and P. Lee. Fault Tolerance: Principles and Practice. Prentice Hall, Englewood Cliffs, NJ, 1981.

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

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

[12] Cristian, F. and S. Mishra. “Automatic Service Availability Management in Asynchronous Distributed Systems,” Proceedings of the 2nd International Workshop on Configurable Distributed Systems, IEEE Computer Society Press, Los Alamitos, CA, March 1992, pp. 58-68.

[13] Cristian, F. “Automatic Reconfiguration in the Presence of Failures,” Software Engineering Journal, Vol. 8 No. 2, March 1993, pp. 53-60.

[14] 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.

[15] 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.

[16] Garlan, D. and D. Perry. “Introduction to the Special Issue on Software Architecture,” IEEE Transactions on Software Engineering, Vol. 21 No. 4, April 1995, pp. 269-274.

[17] 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.

[18] Hofmeister, C. “Dynamic Reconfiguration of Distributed Applications,” Ph.D. Dissertation, Technical Report CS-TR-3210, Department of Computer Science, University of Maryland, January 1994.

[19] Jalote, P. Fault Tolerance in Distributed Systems. Prentice Hall, Englewood Cliffs, NJ, 1994.

[20] 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.

[21] Knight, J. and J. Urquhart. “On the Implementation and Use of Ada on Fault-Tolerant Distributed Systems,” IEEE Transactions on Software Engineering, Vol. SE-13 No. 5, May 1987, pp. 553-563.

[22] 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.

[23] 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.

[24] Laprie, J. “Dependable Computing and Fault Tolerance: Concepts and Terminology,” Digest of Papers FTCS-15: 15th International Symposium on Fault-Tolerant Computing, June 1985, pp. 2-11.

[25] Luckham, D., J. Kenney, L. Augustin, J. Vera, D. Bryan, and W. Mann. “Specification and Analysis of System Architecture Using Rapide,” IEEE Transactions on Software Engineering, Vol. 21 No. 4, April 1995, pp. 336-355.

[26] Maffeis, S. “Electra - Making Distributed Programs Object-Oriented,” Technical Report 93-17, Department of Computer Science, University of Zurich, April 1993.

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

[28] Magee, J., N. Dulay, and J. Kramer. “Structuring Parallel and Distributed Programs,” Software Engineering Journal, Vol. 8 No. 2, March 1993, pp. 73-82.

[29] Magee, J., N. Dulay, and J. Kramer. “A Constructive Development Environment for Parallel and Distributed Programs,” Distributed Systems Engineering Journal, Vol. 1 No. 5, September 1994, pp. 304-312.

[30] 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.

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

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

[33] Moriconi, M. and X. Qian. “Correctness and Composition of Software Architectures,” ACM SIGSOFT Software Engineering Notes, Vol. 19 No. 5, December 1994, pp. 164-174.

[34] 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.

[35] 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.

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

[37] Purtilo, J. and P. Jalote. “An Environment for Developing Fault-Tolerant Software,” IEEE Transactions on Software Engineering, Vol. 17 No. 2, February 1991, pp. 153-159.

[38] 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.

[39] 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.

[40] 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.

[41] Shaw, M., R. DeLine, D. Klein, T. Ross, D. Young, and G. Zelesnik. “Abstractions for Software Architecture and Tools to Support Them,” IEEE Transactions on Software Engineering, Vol. 21 No. 4, April 1995, pp. 314-335.

[42] 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.

[43] Shrivastava, S. and D. McCue. “Structuring Fault-Tolerant Object Systems for Modularity in a Distributed Environment,” IEEE Transactions on Parallel and Distributed Systems, Vol. 5 No. 4, April 1994, pp. 421-432.

[44] Sullivan, K., J. Knight, J. McHugh, X. Du, and S. Geist. “A Framework for Experimental Systems Research in Distributed Survivability Architectures,” Technical Report CS-98-38, Department of Computer Science, University of Virginia, December 1998.

[45] 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.

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

[47] 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.

[48] 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.

[49] 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.