

Survivability Architectures: Issues and Approaches

John C. Knight, Kevin J. Sullivan, Matthew C. Elder, Chenxi Wang
Department of Computer Science
University of Virginia
Charlottesville, Virginia 22903
knight | sullivan | mce7e | cw2e@cs.virginia.edu

Abstract

Survivability architectures enhance the survivability of critical information systems by providing a mechanism that allows the detection and treatment of various types of faults. In this paper, we discuss four of the issues that arise in the development of such architectures and summarize approaches that we are developing for their solution.

1. Introduction

Dependence on large infrastructure systems has increased to a point where the loss of the services that they provide is extremely disruptive. Systems such as transportation, telecommunications, power distribution, and financial services are 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 underlying information systems. The information systems are typically networks with large numbers of heterogeneous nodes that are distributed over wide geographic areas and that employ commodity hardware, and COTS and legacy software. Damage to the information system will in many cases lead quickly to the loss of at least a large part of the service provided by the infrastructure application. We refer to such information systems as *critical information systems*.

Having to deal with events in information systems that might disrupt service leads to the notion of *survivability*. Informally by survivability we mean the ability of the system to continue to provide service (possibly degraded) when various changes occur in the system or operating environment. For example, when events 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 event.

Survivability is a system *requirement*. It is a statement of the required responses to a variety of prescribed events that might affect the system and that might cause a loss of service if nothing is done about them. 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 sufficiently robust that they are largely unaffected by the majority of expected events or that expected events occur with a negligible frequency. Thus, for example, by careful component selection it might be possible to reduce the rate of hardware failures to a negligible level, and by suitably restricting system access it might be possible to eliminate certain types of security attacks.

The informal notion of an “event” that we have used is what is referred to formally in the literature as a *fault* [15]. 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 is *fault tolerance*. This paper is about some of the issues that arise and approaches that we are developing to enhance the survivability of critical information system through the introduction of fault tolerance.

We are concerned in this research with the need to tolerate faults that affect multiple nodes, faults that we refer to as *non-local*. Thus, for example, a widespread power failure in which many network nodes are forced to terminate operation is a non-local fault. Such 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. 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. If necessary, more sophisticated techniques such as virtual synchrony can be used to ensure that the application is unaffected by

local failures. It is important to note that economic factors sometimes force poor technical choices in the mechanisms provided to tolerate local faults thereby causing local faults to be more disruptive than they need to be.

In practice, the approaches that can be followed to enhance survivability are limited. For example, there is little point in considering completely rewriting the software for an existing critical information system because the system is almost certainly too large. Similarly, it is not possible to make drastic changes to the present system architectures. Computers and network links are performing various application functions, and this fabric is determined largely by the application itself. It is not subject to change, at least not in anything but the very long term.

Our approach to tolerating non-local faults and thereby enhancing survivability is through *survivability architectures*. By a survivability architecture we mean a system architecture that is designed specifically to deal with certain non-local faults. In this paper we review four of the issues that occur in the development of survivability architectures and discuss approaches to their solution. The specific topics that we summarize in the remaining sections of this paper are: (1) the form of and rationale for the overall architectural approach that we are developing; (2) an approach to application-system design that permits reconfiguration of an application following an error; (3) mechanisms for achieving security of survivability mechanisms; and (4) an approach to experimentation with new survivability architecture ideas.

2. Survivability Architectures

2.1. Survivability as control

When affected by a non-local fault, a critical information system must be adapted so as to continue to provide information services on which infrastructure service depends. An important issue, therefore, is how this can be done.

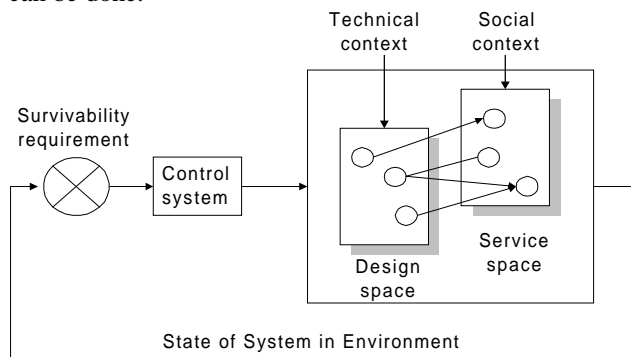


Figure 1. Survivability control system concept

The traditional approach to maintaining service in complex systems (such as avionics platforms) is the use of

explicit control. Our approach to survivability architectures is based on the use of explicit control to manage information systems using data from infrastructures, their information systems, and their operational environments. In essence, such a control system is responsible for choosing a configuration for the critical information system at each point in time based on current conditions to minimize the loss of service, with assurances that under defined circumstances service will meet the survivability requirements.

As illustrated in Figure 1, a given system configuration supports some level of infrastructure service. The need for any given level of service is determined by the system's owner and reflects the owner's assessment of the value of services. The survivability specification documents the service requirements that need to be met for the various anticipated faults. The design space determines the extent to which a control system can manage loss of services when faults manifest themselves. An information system for a *survivable* infrastructure system must have a configuration enabling service provision that meets service requirements for each defined fault to which the infrastructure and its information system is subject.

2.2. Hierarchical adaptive control

The architectural style that we are developing is documented elsewhere [19]. The key characteristics of the architectural style are that it effects *adaptive control* and is *decentralized, hierarchical, and discrete-state*. A control system manipulates a controlled system on the basis of sensor data from the controlled system, predictions of its behavior, and other such information to maintain acceptable levels of system operation. Examples are familiar to every engineer.

A *decentralized* control system is one in which parts of the control system control parts of the underlying system autonomously. An *adaptive* control system is one that can continue providing control in the face of changes to the controlled system and to the control system. For example, an adaptive control system for an avionics application can ensure that an aircraft remains under control even if part of a wing is damaged in flight. A *hierarchical* control system is one in which control actions are determined at a number of levels in a hierarchical system, with low-level control system elements influencing and being influenced by higher levels of control. Tactical decisions might be made close to individual components in a controlled system, while strategic decisions are made at a higher level based on aggregated global system state.

For our purposes, the controlled system is the information system supporting an infrastructure system. In the case of freight rail, for example, the physical system comprises rails, cars and locomotives. This infrastructure is controlled by a complex information system that manages train assembly, dispatch, scheduling, move

authority, billing and so on to meet performance, safety, business and other objectives. A survivability architecture would supplement the information system with a survivability control system. A wide variety of survivability properties can result from the non-local fault tolerance achieved by this structure including intrusion monitoring and response and controlled service degradation under adverse conditions.

The need for a hierarchical structure is implied by the size and distribution of infrastructure information systems. It is implausible, for example, to have a single computing node monitoring the entire United States banking system. Each major bank would have a local control system interacting through abstract interfaces with higher-level (e.g., Federal Reserve) and lower-level (e.g., branch) control systems.

A hierarchical structure is natural to support scalability through local control and the passing of aggregated status information up and down a hierarchy. Such information flows will be needed in practice to implement non-local reconfiguration policies with acceptable performance. Such a structure enables local control nodes to implement policies based on local information and aggregated global state passed from above. In addition to performance, hierarchy enables abstraction and complexity control in control system implementation. Details of local application nodes are abstracted by local control nodes. Higher-level control nodes are specified and implemented in terms of the observable and controllable aspects of control nodes at the next level down the control hierarchy. Hierarchy is also intended to foster evolvability of the control policies. Such evolvability will be critical to effective “learning” by a system over time and as the underlying information and infrastructure systems evolve.

A disciplined approach to the modular design of the control system will also be critical in building *adaptive* control systems that can tolerate the loss, addition, or modification of control and controlled nodes. The control theoretic notion of *multiple-model adaptive control*—in which the control system views the controlled system as being in one of a number of possible distinct operating regimes in which distinct control rules apply—offers especially attractive prospects. Our experimentation system (described in section 5) provides a monitoring capability that is used to connect control nodes in such a way as to ensure that nodes within the control system have a model of both the controlled and control system.

2.3. Implications of survivability as control

Analysis of the concept of survivability as control has revealed the following implications:

- *Application reconfiguration*

For an information system to be subject to control, so as to ensure continued provision of the information services on which an enterprise depends, the

information system must be designed for reconfiguration. That is, the application must provide a sufficiently rich design space to provide scope for a control system to reconfigure it to handle specified adverse conditions.

How best to determine and specify requirements for such flexibility is not clear. It requires an understanding of the impact on customers of service stream interruptions, how information system failures can cause interruptions, and how faults in information systems lead to failures. Moreover, the costs of such flexibility have to be balanced against benefits, the latter of which, like insurance policies, are contingent on the flexibility being needed at some time. A summary of our work in this area is presented in section 3.

- *Design flexibility*

While the analysis and specification of flexibility requirements appear to present significant challenges, implementing the requirements presents additional difficulties. One especially difficult problem is presented by legacy infrastructure information systems. Legacy software systems are an essential part of most infrastructures. The problem is two-fold. First, these systems were most likely not designed to have the kinds of flexibility needed in the face of novel threats. In our domain analysis of several applications we have observed such cases. Second, these systems are generally old, complex, and structurally degraded, and thus hard and costly to change—often impossibly so because they are under tight monetary and intellectual capital-budgeting constraints.

One partial answer appears to lie in transparent extension of the design space of existing systems. In a sense, our recipe for survivability hardening of legacy infrastructure information systems is first to extend (and perhaps also restrict) their design spaces using a wrapping technique; then subject the modified systems to survivability control.

- *Securing survivability mechanisms*

Adding complexity to a complex system in an attempt to make it better often makes it worse. This principle applies to our approach very clearly. A design that inserts into a critical system a control system able to manipulate it in dramatic ways presents an obvious risk: the control system becomes a rich target for a potential adversary.

We believe that, in general, there is no solution to this security problem as we have formulated it. However, techniques can be used to raise the cost of attack to a discouraging level. In practice, a broad range of security and other measures would be taken to provide defense in depth of such a control system. A summary of our work in this area is presented in section 4.

Finally in this section we note that there is likely no single architecture for survivability control. Rather, we

envision an *architectural style* for survivability control based on concepts and structures from the intellectual discipline of control theory.

2.4. Previous work

The application of control systems concepts in software design is not new. Jehuda and Israeli [13] proposed a control system for dynamically adapting a software configuration to accommodate varying runtime circumstances impacting on real-time performance. In CHAOS [10], real-time systems are adapted with the use of an entity-relation database modeling system structure. Control systems ideas have been used in distributed application management as well. Meta [16] is an architecture and a tool that uses a non-hierarchical control system to optimize performance in fault-tolerant distributed systems using Isis. Distributed application management (e.g., [2], [21]) employs services supporting the dynamic management of distributed applications. Network management uses control concepts to manage networks and their running software [3], [6].

However, the major objective in such work is to monitor and improve application or network performance in traditional dimensions, e.g., runtime efficiency. By contrast, our use of control is targeted at enhancing the survivability of controlled applications.

Intrusion detection provides a way to monitor and control the abnormal behaviors of a system. EMERALD [17] introduces an approach to network surveillance, attack isolation, and automated response. It uses distributed, independently tunable surveillance and response monitors as the building blocks, and combines signature analysis with statistical profiling to provide localized protection. A recursive framework is proposed for coordinating the dissemination of analyses from the distributed monitors to provide a global detection and response capability. We address disturbances not limited to security.

GrIDS [18] is a graph-based large network intrusion detection system. It collects data about computer activity and network traffic, and aggregates this information into activity graphs which reveal the causal structure of network activity. This is an intrusion detection system. No response mechanism is discussed. The graph-based detection mechanism could perhaps be used in our architecture.

The Dynamic, Cooperating Boundary Controllers program [22] is developing a capability to allow traditionally static and standalone network boundary controllers (e.g. filtering routers and firewalls) to work cooperatively to protect networks. The capability is achieved through the use of an Intruder Detection and Isolation Protocol (IDIP). The work attempts to address the network intrusion problem only.

Hiltunen and Schlichting propose a model for adaptive systems [11] that respond to changes in three phases: change detection, agreement, and action. It is used for performance and fault-tolerance. Goldberg et al. discuss adaptive fault-resistant systems and present some examples [9]. Our approach provides a way to embed adaptation in the system through multiple model control. Different control policies may be adaptively used for different operating regimes.

3. Application Error Recovery

Tolerating a fault whose effects cannot be masked requires that the application be reconfigured following error detection. Unless provision for reconfiguration is made in the design of the application, reconfiguration will be ad hoc at best and impossible at worst [14].

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 must not introduce new security vulnerabilities in addition to those already noted.

The high-level concept that we are developing is a flexible application architecture based on: (1) the use of formal specification to define the application reconfiguration elements of the survivability requirements; (2) a system architecture that provides critical services to support the application in a *coordinated recovery layer*; and (3) a building block for applications that we term a *reconfigurable process*.

We begin this section 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.

3.1. 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 reconfiguration requirements mean that a solution approach that depends upon traditional software 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 approach 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.

There are many advantages to working with specifications rather than implementations. First and foremost is the ability to specify solutions at a high level. This permits the details of the large number of nodes, the many different node types, and the many different services to be abstracted away to some extent. An implementation-based solution would require too much effort dealing with such a wide variety of nodes, applications, errors, and recovery strategies. In addition, specifications provide the ability to reason about and analyze solutions at a higher level. Finally, an implementation can be synthesized from a specification and this allows recovery strategies to be changed quickly. Different error recovery schemes can be prototyped and explored rapidly.

Precise specification of the error recovery in a critical information system is a complex undertaking. Our approach involves implementation synthesis by a translator from three major sub-specifications:

- *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.
- *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).

The overall structure of the specification is that of a finite-state machine that characterizes the requisite responses to each fault. Arcs are labeled 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 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. 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.

3.2. Reconfigurable processes

We define a specialized type of application process, the *reconfigurable process*, which 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 recoverable 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 critical services that a reconfigurable process has to support include:

- Start, suspend, resume, terminate, and delay.
- Change process priority.
- Report prescribed status information.
- Establish recovery point, and 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 services 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 that manipulate databases. Such processes have to be very carefully developed if the creation of a checkpoint is to be efficient.

3.3. Node architecture

Each node in a critical information system that supports comprehensive error recovery using the approach discussed here will have an architecture that complies with certain constraints. 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. As noted in section 2.3, for legacy systems achieving this architecture will probably require careful wrapping of some components.

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 discussed 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 actuator 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 survivability specification translator synthesizes the actuator implementation.

3.4. 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.

The *coordinated recovery layer* provides the interface that is used in the implementation of critical services within reconfigurable processes. The following is a list of the functions that the coordinated 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 coordinated recovery layer provides these services in a largely application-independent manner. Thus, a common coordinated 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.

4. Securing the Survivability Mechanism

The survivability architectures that we are developing using the mechanisms described in the previous two sections have the unfortunate characteristic that their corruption could result in serious security intrusions or failures in the system's ability to provide service. For example, if the sensory component of a network control system is compromised, it effectively cripples the mechanism's ability to monitor the system. Similarly, if the actuator mechanism were penetrated, the intruder could gain access to the control of the entire network simply by manipulating the error-recovery controls.

Such subversions are much more dangerous than a typical security intrusion. In a typical intrusion, the intruder might be able to gain unauthorized access to a part of the system (such as a single node) but compromise of the entire system is unlikely. Despite this, many current network management schemes and intrusion-detection systems¹ do not address the security and reliability issues of the mechanism that is being employed, even when executing on COTS platforms that are of questionable trustworthiness. The underlying assumption is that these

¹ Intrusion detection systems are special cases of network monitoring systems and are a useful example of the system security issues.

mechanisms will be secure against malicious attacks. Given the types of system within which these mechanisms are deployed, this assumption is unfounded and misleading to say the least.

We argue that protection of the survivability mechanism is of paramount importance, and that the analysis and treatment of this problem should be brought to the foreground of research—as a minimum, the consequences of security failures in the survivability components must be studied and understood. Until the limitations of software protection are well understood, the current paradigm of intrusion detection and network management is inherently dangerous.

In this section we discuss the basic problems in preserving execution integrity of software in untrustworthy environments, and summarize our solution approach.

4.1. Vulnerabilities in the monitoring-and-response paradigm

The task of *monitoring* is to collect information from the target system. It is essential that this process be dependable because it constitutes the foundation of subsequent analysis. The task of *response* is to take the results of analysis and make any changes to the network that are implied by the analysis. It is essential that this process be dependable because it constitutes the foundation of the ensuing network configuration.

A generic survivability mechanism includes a sensing component, an analysis component and an actuating component. It must be assumed that the application hosts are vulnerable to security attacks (hence, in part, the reason for monitoring). Because of the security implications of failures in monitoring and response, it must also be assumed that each of the components of the survivability mechanism will be attractive targets for attackers, and that direct corruption of the monitoring and response process is a distinct possibility.

If the program or data of the sensing component is corrupted, the perpetrator could, as a minimum, cause denial of sensing services, i.e. the sensor no longer executes. A more devious attack is to corrupt the sensors in some specific way such that the changes are undetected, and the sensor will go on functioning on corrupted states or data.

Attacks that entail sophisticated tampering with or impersonation of the sensor program or data are the most dangerous. Consider a networked environment where monitoring information is being collected from distributed locations across the system. A carefully coordinated attack on a selected set of sensors could cause the analysis to reach an inaccurate view of the state of the network, and arrive at erroneous reactive decisions that may lead to further deterioration of the system services.

Security attacks against the actuators could be far more serious. If the actuators are compromised, the intruder will have available a powerful tool to affect the *entire* system state and thereby to perform malicious actions. For example, malicious commands to the actuators could be issued that caused nodes to shut down, data to be corrupted, operating modes to be switched, or any similar disruptive event.

The security implications of the monitoring-and-response paradigm for survivability in critical information systems are significant. This problem is especially troublesome in the management of large network systems. First, the scale and size of the systems suggest likely heterogeneity in the technology and administration employed in the system—some sites will be more easily penetrated than others and securing each individual host from the ground up is clearly not an option. Second, in order to make timely non-local management decisions, it is necessary to correlate and integrate local information. This implies complex control mechanisms as well as inherent difficulties in identifying corrupted security components.

4.2. A security architecture

Mindful of the problem description described above, in this section we present an overview of our solution approach.

Wherever possible, traditional security mechanisms can be applied. For example, isolation of the survivability mechanism is the starting point that we adopt. If the survivability mechanism were isolated, then extreme physical security and strict access controls could be used to ensure secure operations.

However, complete isolation of the entire survivability mechanism is impossible. The survivability mechanism has to have access to the application system to both monitor and respond. A recurring theme is that parts of the survivability mechanism, i.e., the very system element that detects errors (e.g., discovers security attacks) and attempts error recovery, must execute in an *untrustworthy* environment. The fact that various components of the survivability mechanism itself could be the target of security attacks raises the following questions:

- To what extent can this vulnerability be minimized by isolation?
- To what extent can the elements of the survivability mechanism that cannot be isolated be protected?

The analysis component of the survivability mechanism can be isolated as can communication between the analysis component and the sensing and actuating components. In fact, everything but the initial data collection performed by the sensing component and the final action interface between the actuating component and the application can be isolated. Thus, the answer to the first question is quite straightforward—almost all of the survivability mechanism can be isolated. Unfortunately,

what remains is crucial, and the serious vulnerabilities outlined above remain. We turn now to the second question and note that traditional techniques such as encryption cannot be used for this particular problem because in general code cannot be encrypted and because the data has to reside in an untrustworthy environment where keys cannot be protected.

Our approach to securing the sensors and actuators is based on two concepts: *one-way translation* and the extensive use of *diversity*. By one-way translation we mean a compilation process that generates executable programs that operate normally but which cannot be easily reverse engineered to permit recreation of the source program. By diversity we mean the use of components that differ in some crucial aspect but are otherwise similar.

4.2.1. One-way translation. The goal with one-way translation is to maximize the difficulty experienced in trying to discover details of the source programs that could have produced a given executable program when starting with the executable program. That is, transformations are applied at various stages of compilation to introduce differences between the program's source and binary representations. If there are sufficient differences, then no mapping from an executable program back to its original source program can be established in a prescribed time frame.

The role of one-way translation in securing the survivability mechanism is to allow the executable form of the sensing and actuating software to reside on untrustworthy hosts yet for compromise not to be possible in bounded time. Thus if an adversary is able to penetrate the application node and copy the executable programs, he or she will not be able to discover the information needed to corrupt those programs within some prescribed time limit. For example, if an adversary were to obtain the source program for the sensing component and were able to copy the executing program, he or she would have to be able to locate specific buffers and code sequences in memory in order to attack the control system via the sensor. With one way translation, the claim is that this could be made sufficiently difficult that it could not be completed within some time bound.

A wide variety of techniques exist to introduce complexity and variability into executable programs, some of which are discussed in other studies [1] [4] [5] [12]. A critical problem is that the introduction of variability that is intended to reduce the possibility of analysis of the resulting executable program must be shown to actually achieve that reduction. Ad hoc techniques that "seem like they ought to work" are not sufficient.

Some example semantics-preserving transformations that can be applied include:

- *Random placement*
The compiler uses a randomized allocation algorithm when generating addresses for variables and code.

- *Random code selection*
The compiler selects at random from different code skeletons for the same source construct.
- *Execution-time reordering*
The execution-time system relocates code and data at random times.
- *Introducing concurrency*
Parts of a program that are normally sequential are transformed into a concurrent form.
- *Introducing aliases*
Aliases to variables and functions are introduced so that access to a single address can always be achieved through any one of multiple pointers.

Each of the above can play a role in achieving one-way translation. For each it is necessary to develop a model that predicts the difficulty that the specific technique introduces into the goal of defeating reverse engineering, and this raises the question of how such modeling is done.

To illustrate the analysis, consider the last technique listed—introducing aliases. Detection of aliases by static analysis is known to be NP complete. By introducing it deliberately into a program during compilation, and to do so in large quantities, leaves an executable program that in general cannot be analyzed without the expenditure of unreasonable resources. In practice, additional work is required to determine circumstances under which a useful lower bound exists on the resources required for static analysis.

To assess the performance of this (and other) techniques, a source-to-source translator for C is being developed. The translator introduces aliases at the source level and the resulting program is then compiled normally.

4.2.2. Diversity. Diversity is an important engineering technique in building dependable systems. For example, in the design of an aircraft, spatial diversity is used in the layout of hydraulic lines—each of the three redundant hydraulic lines feeding control surfaces pass through different parts of the fuselage and wings. This design helps to ensure dependable operation by tolerating some perturbations in the environment.

Incorporating diversity into the design of secure systems helps to reduce vulnerabilities that arise from uniform designs that include replicated flaws [8]. We identify two forms of diversity—*spatial* diversity and *temporal* diversity—that are beneficial in securing software execution:

- *Spatial Diversity*
By spatial diversity we mean the deployment of diverse software copies of security components over different locations. This will mean that if one copy of the component is compromised, the same attack might not work on other components and that they will remain functional. Spatial diversity is especially important given that a large number of known security attacks are based on exploitation of common software

and configuration flaws.

- *Temporal Diversity*

By temporal diversity we mean the deployment of diverse software copies of security components over time. This will mean that effort expended by an attacker to penetrate a component will be ineffective provided the component is changed before the attack succeeds.

As an example, consider a survivability control system in which the sensors are communicating status information to the analysis engine, and assume that the status information is encoded into a set of numerical representations. If the mapping from status to numerical representation differs from node to node, an observer watching messages going across the network will not be able to deduce easily what status is being reported. Even if the adversary successfully compromises one mapping and is then able to impersonate that particular sensor, he or she will not be able to impersonate other sensors using the same information.

Despite one-way translation and spatial diversity, an intruder might eventually be able to deduce what the program is doing and corrupt or impersonate the legitimate sensors and actuators. This is analogous to the possibility that encrypted information might eventually be revealed or a password discovered by a brute-force attack. And in the same way that information privacy is strengthened by carefully changing encryption keys and passwords, control system security can be improved by introducing temporal diversity.

Temporal diversity in this case amounts to replacement of the system elements that might have been compromised with new versions based on different randomization. Thus, for example, sensor and actuator binary programs can be replaced with new instances that have been compiled using different randomization keys in the one-way translation process.

A technical challenge with such a mechanism is in the seamless transition of one version of the executing program to another. Since program replacement can happen at arbitrary points during execution, it must be possible to dynamically save and restore state information. Several techniques for state save and restore have been proposed in the literature. The prominent ones include Ferrari's Process Introspection [7] and Migration by Recompile [20], both of which involve the notion of dynamic checkpointing.

Furthermore, dynamic replacement of executable programs must be a secure operation in the sense that there must be a trustworthy path from the generation to the delivery of the binary program. A careful authentication mechanism needs to be in place for the monitored hosts to authenticate the origin of the program binaries—it must only originate from trusted sources, otherwise the recharge mechanism will be itself a vulnerability that can be exploited. Signing each binary with the trusted generator's private key is a good start.

5. Experimentation System

Experimentation is a crucial element of research in survivability. It is vital, for example, to the early evaluation of novel survivability techniques. Unfortunately, several factors present serious impediments to experimentation. First, infrastructure systems are privately owned. Second, they are, by definition, critical from both business and societal perspectives. It is inconceivable that their operators would permit experimentation on them. Third, infrastructures are enormous in physical scale, cost and complexity, and this makes it infeasible to replicate them in the laboratory.

We have thus adopted an approach based on the use of *operational models* of infrastructure systems as test-beds for developing and evaluating prototype architectural mechanisms for survivability. In this section we present an overview of an experimentation system that permits a wide range of representative models of critical infrastructure systems to be built rapidly and made the subject of experimentation. Provision is made within the experimentation system for creation, manipulation, and observation of models of infrastructure systems as well as the introduction of architectural elements designed to enhance survivability. The ability to develop and analyze models and prototype survivability mechanisms rapidly is an important aspect of the work because we wish to explore a range of infrastructure applications and a variety of architectural concepts efficiently.

Given the impediments to direct experimentation with real infrastructures, we have adopted an experimental approach based on operational models. By an *operational model*, we mean a simplified version of the real system that executes as the real system does—as a true distributed system—but that provides only a restricted form of its functionality. The goal for a given model is to have a simplified laboratory version of the associated infrastructure system that is made manageable by implementing only relevant application functionality and implementing only essential characteristics of the underlying target architecture. Most infrastructure applications, for example, are distributed and this is an essential characteristic, although in most cases the particular protocols used in the underlying network are not. Thus for our purposes an operational model needs to be truly distributed but it need not use any specific network protocol.

Building operational models of critical infrastructure information systems presents two significant challenges: (1) modeling the critical and no other aspects of infrastructure systems with sufficient accuracy and completeness; and (2) facilitating inclusion in the model of relevant architectural mechanisms to be developed or evaluated. For purposes of experimentation, an operational model has to represent relevant functional and architectural features of a given system, as well as its

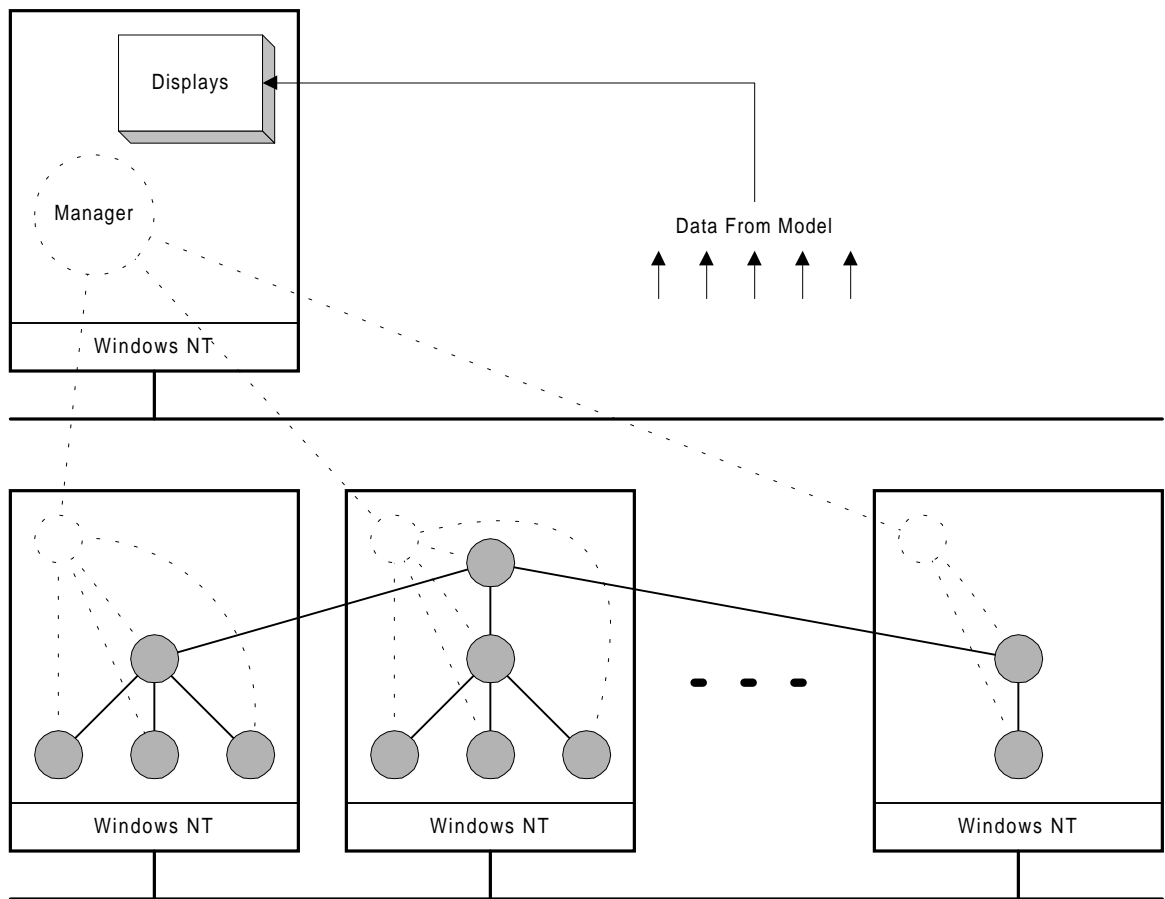


Figure 2. Experimentation system

operational environment, including a dynamic model of internal failures and external threats. Once such a model is built, mechanisms must be present to allow prototypes of architectural survivability mechanisms to be introduced. Both models and architectural supplements must be instrumented for collection of data needed to analyze and evaluate survivability mechanisms.

Since for our purposes an operational model is a distributed system, the experimentation system implements distributed systems with arbitrary numbers of nodes and arbitrary interconnections between nodes. The basic communications paradigm that the experimentation system provides is message passing. Thus, an operational model within the described experimentation system is a collection of concurrent computational nodes between which messages of arbitrary complexity can be passed. In reality, the nodes and their communication are provided by a set of networked Windows NT workstations with a

prescribed association between nodes in the model and physical computers.

Figure 2 provides an overview of the experimentation system. Solid rectangles represent physical computers; gray circles represent nodes in the model; and solid lines between gray circles represent communications links. Small white circles are *proxy* processes that act on behalf of the *manager*, represented by the large white circle. The manager provides an interface to control individual nodes remotely so that, for example, nodes can be managed on physically remote workstations. Each workstation executes a *proxy* process to implement management functions for the nodes on that workstation. "Data From Model" is a mechanism for collecting data about the operating model for display.

For purposes of experimentation, our system provides the user with an efficient, easily manipulated operational model of a distributed application with extensive control,

monitoring, and display facilities. It also includes mechanisms for modifying the architecture of the system (see below) to permit experimentation with a wide variety of architectural concepts.

5.1. Building block for models

The experimentation system provides building blocks that permit models to be built, executed with typical data streams, observed, and controlled. The basic building block (i.e., the gray nodes in Figure 2) is an executable Windows NT process that we refer to as a *virtual message processor*. A virtual message processor provide two basic functions:

- A mechanism for interpreting messages.
- A mechanism for sending and receiving messages to and from other virtual message processors.

Figure 3 depicts the general structure of a virtual message processor. A virtual message processor maintains a queue of incoming messages and a queue of outgoing messages. Incoming messages are routed from the input queue to programmable message interpreters. Any new messages generated as a result of interpreting a received message are placed in the output queue. Messages can be generated asynchronously also if needed based on, for

example, a timer event.

A requirement that is especially difficult to satisfy is the need to model both normal and alternate application functionality. Clearly, no simple mechanism will allow arbitrary functionality to be modeled. The way that the experimentation system meets this requirement is to allow explicit programming of message interpreters. Message interpreters absorb a message destined for them and effect whatever minimal processing is required to achieve the required level of functionality. Naturally, message interpretation often involves generation of new messages for transmission elsewhere.

Virtual message processors provide network addressing and message transmission at the model level to support required communication structures. To accommodate experimentation, messages to and from virtual message processors can be configured in two ways—with *direct* or *indirect* connection. Other nodes are unaware of how any given node is connected.

With direct connection, the message traffic for any particular virtual message processor is routed to and from destinations and sources normally. With indirect connection, all messages to and from that particular virtual message processor pass through a “wrapper” that is not part of the application functional model. Since the wrapper can contain message interpreters for all messages, it

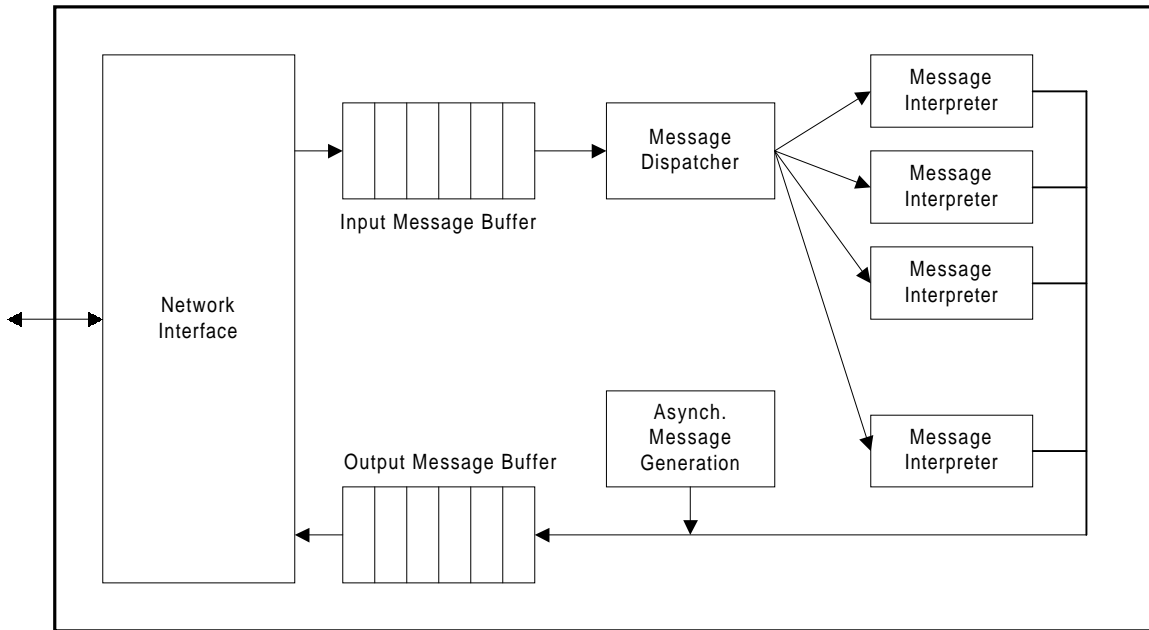


Figure 3. Virtual message processor structure

provides a mechanism for transparent monitoring of all application network traffic in which a virtual message processor engages.

We refer to the interposed wrapper as a *protection shell* since conceptually it is a shell around the target virtual message processor. The target message processor cannot communicate except through the shell. If the shell's message interpreters merely copy received messages to the output, then the shell's presence is entirely transparent. However, since it can "see" all messages and process them at the application semantic level, it can manipulate them in any manner that is required. Minimally, it can monitor message types and content, but more importantly, it can synthesize information about the state of the virtual message processor that it surrounds for use by higher levels of control.

Finally, virtual message processors can transmit messages to their own shells and vice versa (message are just messages) thereby permitting any degree of state communication that is desired. Thus shells can acquire and also change any information about their subject processor's state.

5.2. Building operational models

Models are built by combining virtual message processors in various ways. A model of a distributed infrastructure application could be built using a virtual message processor for each application node in the network. The network topology is defined in a file using a simple format that defines node addresses and communication links. Each virtual message processor has either a name or an integer address within the model and network links can be specified between any pair of nodes. Thus arbitrary network topologies are easy to specify, and large models can be defined easily. Models are limited in size mostly by available resources but the task of merely defining a model for a large system is tedious and therefore also a restriction. Our available resources permit the creation of models with up to about 1,500 nodes in the laboratory. Defining such models is simplified by a model synthesis system that creates model descriptions from a small set of model parameters (number of nodes, type of topology required, maximum communication fan out, etc.)

Models can be executed on any available set of workstations running Windows NT. Application nodes and their interconnections in the model are mapped to physical workstations by specifying a mapping from node number to IP address. To enable more effective performance studies of prototype architectural survivability mechanisms, network connections in the model are mapped to efficient low-level socket connections.

Architectural supplements, such as superimposed survivability control systems, are also built for the most part using the same virtual message processors as building

blocks. These components provide the additional computational facilities that the architecture requires. Shells give the ability to insert an additional processing step transparently into an otherwise active link between two virtual message processors. This, coupled with the ability of nodes to communicate by passing arbitrary message types, provides all the necessary facilities for interaction between model components and the supplementary survivability architectural mechanisms to be evaluated.

Modeling the environment requires supporting the application system's input and output requirements and making provision for all types of faults. The virtual message processor construct supports these requirements. Virtual message processors model application input and output by producing and transmitting appropriate messages to application model nodes and by receiving and processing their output messages.

Finally, the virtual message processor construct supports control, display, and failure injection with specialized messages. The manager and proxy processes are merely virtual message processors that effect control by message transmission. Failure injection is implemented by sending messages from the manager to the virtual message processors to be affected, which then interpret the messages suitably. Thus, for example, node failures are effected by the node ceasing to process messages after the failure message is received.

6. An Example System

We illustrate some of the ideas discussed in this paper using a critical application from the banking system (the U.S. financial payment system) as an example. Our selection of this particular application was based on its absolutely crucial national significance—failure would be catastrophic for the nation—and the consequent need to defend it.

The goal for this particular activity was to explore a survivability architecture in which a supplementary hierarchical control system is used to react to widespread failures of several types. This case addresses five critical research issues: (1) the feasibility of distributed, hierarchical control as a survivability architecture; (2) the control algorithms required to respond to failures of different types; (3) mechanisms that could be employed to specify survivability control policies for the control system; (4) the feasibility of detecting the errors resulting from non-local faults; and (5) the utility, flexibility and ease of use of the experimentation system.

6.1. Model architecture

The overall network structure in this model is a tree. This is typical of the way banks are connected for payment purposes, but our topology is strictly hypothetical. The

model includes four types of application node. The first is a *branch* bank providing customer service. Such nodes appear as leaf nodes in the tree. The second type of node is a *money center* bank—essentially the primary information center for a single commercial bank. A money-center bank appears as an intermediate-level node in the tree, and has a set of local banks as children. The third type of node represents the *Federal Reserve System* and is the root of the tree. Money-center banks are connected to the Federal Reserve System. The fourth node is a transparent node that merely acts to connect active nodes.

Needless to say, the information system that effects payment in the United States is a very large network, and we could not model this scale. Our current model is composed of several hundred application nodes with various numbers of money-center and branch banks. There is a single Federal Reserve System.

The model of the Federal Reserve System includes a primary server and a local hot spare that is permanently synchronized with the primary server and is able to mask failures of the primary. We also model a geographically remote warm spare that mirrors the data held by the primary. The warm spare is able to provide service to the remainder of the network if the primary and hot spare fail; but in order for it to do so the money-center banks must reroute payment requests and wait for service to be initialized. This model is representative of the availability mechanisms actually used by the Federal Reserve System.

6.2. Application functionality

The application functionality we have implemented in the model includes check processing and large electronic funds transfers. Each payment demand includes typical routing information—source bank, source account number, and destination account number as well as the payment amount. User's bank accounts are held at the branch banks and it is there that all payment requests are made. A load generator creates random sequences of payment demands that take the form of either a "check" or an EFT request.

As in the real payment system, payment demands below a certain threshold value are grouped together so that funds transfers between money-center banks are handled in bulk by the Federal Reserve System at scheduled settlement times. Bulk funds received by a money-center bank have to be dispersed through the bank's own network so that the correct value reaches each destination account. This part of the application models the processing of paper checks.

Transfers of funds where the value exceeds the threshold value are effected individually and upon receipt of the demand. This aspect of the application models large EFT request processing. The two-tier approach to payment processing is representative of the overall structure of the real payment system.

6.3. Architectural supplement

The control systems architecture for this example is a small, distributed system that is separate from the application system. Protection shells that surround application nodes undertake sensing the state of application nodes and transmitting commands for reconfiguration. To permit reconfiguration to be tailored to different semantic levels in the application network topology, the control system operates hierarchically with lower levels supplying summary information to upper levels to optimize control decisions.

For purposes of experimentation, the model implements two extremely simple survivability policies that are designed only to demonstrate and evaluate the control-based survivability architecture and key aspects of the experimentation system:

- *Comprehensive Shutdown*
This policy requires that the entire payment system be shut down if any application node except the Federal Reserve System fails. This is quite unrealistic of course but its purpose is to permit experimentation with hierarchic control.
- *Federal Reserve Redirection*
This policy requires that the entire payment system switch to the use of the geographically remote warm spare in the Federal Reserve System in the event that the primary server and hot spare both fail.

In both of these policies the notion of failure is quite general. For simplicity, we do not distinguish between types of failure at this point. Thus "failure" might mean physical damage or a security penetration.

None of the key services required by transaction processing systems (such as two-phase commit protocols) are provided by the modeling framework, nor are they intended to be. The modeling of continued service that is present in this example is at the level of system and application management. Important issues such as consistent recovery in distributed heterogeneous systems are abstracted away. The focus is on monitoring and control in large distributed systems. We assume that lower level details are provided by the application. They could be added explicitly as part of the application functionality in a model built for a different research goal.

6.4. Error detection

Separately from the simple survivability policies discussed in the previous section, we have studied the issue of error detection for non-local faults. As an example, we have studied the question of whether the effects of a virus or worm that has infected large parts of a network can be detected. A useful result of the facilities of the experimentation system is that we have complete control over the infection model, the detection semantics within each node, the actual time at which events take

place, and the time at which the control system observes any element of the erroneous state.

The question of interest is whether the presence of an active virus in a network will create a detectable signature.

6.5. Model implementation

The topology of this model is defined entirely in a specification in the experimentation system's configuration file. Application nodes are virtual message processors and the application's communications system is implemented by links between virtual message processors. The control system architecture model is also built with virtual message processors.

The application functionality is implemented by small sections of C++ source providing message interpretation in the application nodes. The functionality implied by the redundancy model for the Federal Reserve System is achieved with a trivial amount of programming within the application functionality.

For purposes of experimentation with this model we defined a set of failure injection messages. These messages can cause any group of nodes to fail either concurrently, in a random pattern over a prescribed time interval, or in a cascading fashion over time in which only nodes connected to a failed node can fail. These failure models provide a basis for research on architectural mechanisms to detect and handle such occurrences in real infrastructure systems.

6.6. Results

Our results obtained with this model are in three areas: (1) the utility of the experimentation system; (2) the performance of the experimentation system; and (3) the feasibility of hierarchic control of network survivability using the control system paradigm. In the first area, the experimentation system supported development of the model that we have described in all respects. Building of the model was simple and its specification is short. The facilities of the model, especially the pattern of use of components, met all of our demands. We have built several versions of the model quickly and incrementally.

To date we have assessed the runtime performance of the experimentation system (as opposed to its support for model construction) informally and only in the area of runtime performance on a physical computer. On a Pentium-based machine with 128 Mbytes of main memory and a typical disk configuration, acceptable performance is obtained with up to about 350 virtual message processors running concurrently processing messages associated with the payment-system model.

The model described here incorporates a preliminary hierarchic control system that is designed to provide significant survivability enhancement. Although no performance quantification has been undertaken, the

model demonstrates the feasibility of non-local state assessment and damage assessment coupled with a hierarchic approach to state restoration, and continued service. The latter is especially important since survivability of large distributed applications will require the following two activities to cope with major failures:

- Significant reconfiguration of the network's topology (state restoration) where different elements of the reconfiguration are coordinated yet tailored to different circumstances throughout the network.
- Substantially different alternative or reduced applications (continued service) at different locations based again on the different circumstances throughout the network.

Finally, our studies of virus propagation in networks have revealed behavior that was quite unexpected. For example, the variation over different runs in the time required to infect the entire network using the same infection model varies by a factor of two in some topologies. This along with related observations of variability in other characteristics suggests that detection of non-local viral attacks might be very difficult.

7. Conclusions

Survivability architectures offer a useful approach to the provision of fault tolerance in critical information systems. In particular they offer an approach to tolerating faults in which the continued service element of fault tolerance differs from normal service, i.e., the effects of the fault are not masked. By introducing this type of fault tolerance, progress can be made towards meeting the survivability goals of critical infrastructure applications.

In practice, implementing survivability architectures raises many issues that have to be dealt with before the techniques can be applied. Particular issues that we have started to address include application reconfiguration and security of the survivability mechanisms.

A significant difficulty arises when the various concepts involved in survivability architectures have to be evaluated because experimentation with real systems is precluded. Our approach to dealing with this problem is to use operational models that can be built and studied in the laboratory using an experimentation system that we have developed.

8. Acknowledgements

It is a pleasure to thank John McHugh, Raymond Lubinsky, and Xing Du for technical discussions and assistance with the work described in this paper. We also thank the anonymous referees for very helpful comments. 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. Kevin Sullivan was also supported by the National Science Foundation under grants CCR-9502029, CCR-9506779, CCR-9804078.

9. References

- [1] D. Aucsmith, "Tamper Resistant Software", *Proceedings: First Information Hiding Workshop*. Cambridge, UK, 1996.
- [2] M. A. Bauer, R. B. Bunt, A. El Rayess, P. J. Finnigan, T. Kunz, H. L. Lutfiyya, A. D. Marshall, P. Martin, G. M. Oster, W. Powley, J. Rolia, D. Taylor, and M. Woodside, "Services Supporting Management of Distributed Applications and Systems", *IBM Systems Journal*, Vol. 36 No. 4, 1997, pp. 508-526.
- [3] B. Boardman, "Network Management Solutions Lack Clear Leader", *Network Computing*, August 15, 1998, pp. 54-67.
- [4] C. Collberg, C. Thomborson, D. Low, "Breaking abstractions and unstructuring data structures", *Proceedings: IEEE International Conference on Computer Languages*, Chicago, May 1998.
- [5] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations", *Technical Report 148, Dep.t of Computer Science*, University of Auckland, July 1997.
- [6] Computer Associates, "Enterprise Management Strategy: Managing the New Enterprise", White paper, <http://www.cai.com/products/unicent/whitepap.htm>, 1996.
- [7] A. Ferrari, "Process State Capture and Recovery in High-Performance Heterogeneous Distributed Computing Systems", *Ph.D. Dissertation*. University of Virginia, January, 1998.
- [8] S. Forrest, A., Somayaji, and D. Ackley, "Building Diverse Computer Systems", *Proceeding: 6th workshop on Hot Topics in Operating Systems*, IEEE Computer Society Press, Los Alamitos, CA, 1997, pp. 67-72.
- [9] J. Goldberg, L. Gong, I. Greenberg, R. Clark, E. D. Jensen, K. Kim, and D. Wells, "Adaptive Fault-Resistant Systems", *Technical Report*, SRI International, 1994.
- [10] P. Gopinath, R. Ramnath, and K. Schwan, "Database Design for Real-Time Adaptations", *Journal of Systems and Software*, Vol. 17, 1992, pp. 155-167.
- [11] M. A. Hiltunen and R. D. Schlichting, "Adaptive Distributed and Fault-Tolerant Systems", *International Journal of Computer Systems and Engineering*, Vol. 11 No. 5, 1995, pp. 125-133.
- [12] F. Holhe, "Time limited black box security", *Lecture Notes of Computer Science, Vol 1419. Mobile Agents*. Springer-Verlag, 1998.
- [13] J. Jehuda and A. Israeli, "Automated Meta-Control for Adaptive Real-Time Software", *Real-Time Systems*, Vol. 14, 1998, pp. 107-134.
- [14] J. Knight, M. Elder, and X. Du, "Error Recovery in Critical Infrastructure Systems", *Proceedings: Computer Security, Dependability, and Assurance Workshops 1998*, IEEE Computer Society Press, Los Alamitos, CA, 1999, pp. 49-71.
- [15] J. Laprie, "Dependable Computing and Fault Tolerance: Concepts and Terminology", *Digest of Papers FTCS-15: 15th International Symposium on Fault-Tolerant Computing*, IEEE Computer Society Press, Los Alamitos, CA, 1985, pp. 2-11.
- [16] K. Marzullo, R. Cooper, M. D. Wood, and K. P. Birman, "Tools for Distributed Application Management", *IEEE Computer*, August 1991, pp. 42-51.
- [17] P. A. Porras and P. Neumann, "EMERALD: Event Monitoring Enabling Responses to Anomalous Live Disturbances", *Proceedings: 20th National Information Systems Security Conference*, October 7-10, 1997.
- [18] S. Staniford-Chen, S. Cheung, R. Crawford, M. Dilger, J. Frank, J. Hoagland, K. Levitt, C. Wee, R. Yip, and D. Zerkle, "GrIDS - A Graph Based Intrusion Detection System for Large Networks", *Proceedings: 19th National Information Systems Security Conference*, October 22-25, 1996, pp. 361-370.
- [19] K. Sullivan, J. Knight, X. Du, and S. Geist, "Information Survivability Control Systems", *Proceedings: 21st International Conference on Software Engineering*, IEEE Computer Society Press, Los Alamitos, CA, 1999, pp. 49-71.
- [20] M. Theimer, and B. Hayes. "Heterogeneous Process Migration by Recomilation", *Proceedings: 11th International Conference on Distributed Computing Systems*, IEEE Computer Society Press, Los Alamitos, CA, May 1991.
- [21] Tivoli Systems, "Tivoli and Application Management", White paper, http://www.tivoli.com/o_products/html/body_map_wp.html, 1998.
- [22] UC Davis and Boeing Co., "Intrusion Detection and Isolation Protocol (IDIP)", <http://www.cs.ucdavis.edu/projects/idip.html>.