Architectural Approaches to Information Survivability

John C. Knight
Dept. of Comp. Sci.
University of Virginia
Charlottesville
VA 22903, USA
+1 804 982-2216
knight@virginia.edu

Ray W. Lubinsky
Dept. of Comp. Sci.
University of Virginia
Charlottesville
VA 22903, USA
+1 804 982-2258
rwl@virginia.edu

John McHugh
Dept. of Comp. Sci.
Portland State Univ.
Portland
OR, USA
+1 503 725-5842
mchugh@cs.pdx.edu

Kevin J. Sullivan
Dept. of Comp. Sci.
University of Virginia
Charlottesville
VA 22903, USA
+1 804 982-2206
sullivan@virginia.edu

ABSTRACT
Many large information systems have evolved to a point where the normal activities of society depend upon their continued operation. Significant concerns have been raised about the possible effects of failure in these systems.

In this paper we discuss architectural approaches to improving the survivability of critical information systems and present a candidate architecture. The key features of the architecture are the use of a variety of shell structures (sometimes also known as wrappers) and the use of a network-wide approach to recovery and continued service. We discuss the design, implementation, and verification issues raised by the use of shells in complex distributed systems and introduce three types of shell: protection, enhancement, and correction. Combinations of these shells are used to ensure that the critical information system is protected against a wide variety of hazards ranging from software defects to malicious attacks.

The implementation of shells is discussed and it is shown that the desirable characteristic of transparent implementation cannot generally be achieved, and that ensuring the correct operation of the shells is itself a significant issue. A demonstration system being developed for evaluation of the architectural concepts is presented.

Keywords
Survivability, critical infrastructure applications

1 INTRODUCTION

Many large information systems have evolved to a point where organizations rely heavily upon them. In some cases, such systems are so widespread and so important that the normal activities of society depend upon their continued operation; management of transportation systems such as air traffic control, telecommunications, nationwide control of power distribution, and the financial system are examples. We refer to such systems as *critical information systems*.

Societal dependence on these systems is growing and will continue to do so for the foreseeable future as large amounts of inexpensive computing and network hardware become available. This new hardware, particularly the communications capability that it provides, offers the opportunity to enhance existing applications in innovative ways and develop new applications. We note, however, that new applications are usually constructed in part from existing components and are frequently required to inter-operate with existing applications.

Complicating the situation is the interdependence of some of these applications. For example, although limited protection against loss of power is afforded for some information systems, service following a power loss is usually severely reduced. Thus, for example, management of transportation systems will be affected significantly if there is a widespread loss of power. Similarly, loss of communication service will disrupt many other information systems such as finance, electronic commerce, and transportation.

The dependability requirements that arise with many critical information systems are quite extraordinary. For example, many current systems and others that are being planned are required to operate on networks that are distributed nationally (sometimes globally) and require twenty-four-hour-per-day, seven-day-per-week operation. In addition, these systems have to support combinations of dependability requirements. For example, they have to maintain very high levels of availability whilst also ensuring network-wide security.

The loss of the services that these systems provide could be very serious. Some of the consequences of failure are fairly obvious—the failure of part of the air traffic control system has manifest implications. Other consequences of failure are less obvious. The failure of certain parts of the banking system, for example, can have widespread negative impact and do so very quickly. A number of failures have already been reported [14].

Although modern information systems provide excellent service for the most part, concerns have been raised about the possible effects of failure in critical information systems [1, 3, 8]. These concerns have been heightened by the growing societal dependence on these systems, and by their increasing number and complexity.

Dealing with the effects of faults in information systems 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 operating environment. For example, when events such as 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.

There is a need to improve the survivability of critical information systems given the increasing dependence on them, the serious consequences of their failure, and their demonstrated fragility and vulnerability [1, 8]. However, the approaches that can be followed to achieve this goal are limited. For example, there is little point in considering completely rewriting the software for the systems because they are just 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.

In this paper we discuss architectural approaches to the provision of survivability with particular emphasis on dealing with software and security problems. We introduce a novel architecture that permits individual nodes to be strengthened without disrupting the application design. A key feature of the architecture is the provision of facilities to manage network-wide error detection and recovery algorithms.

2 THREATS AND VULNERABILITIES

Large information systems fail for many reasons. Thus, in order for a critical information system to survive, there is a wide variety of faults whose effects have to be dealt with explicitly when they occur, i.e., faults that have to be tolerated. They include the following:

- hardware degradation faults,
- hardware design faults,
- software faults,
- faults in operational procedures,
- faulty actions by operators,
- changes in physical environmental conditions, and
- malicious attacks.

Techniques exist for dealing with many of these types of faults. Redundant components, for example, can be configured to deal acceptably with hardware degradation faults, and geographic diversity of replicated entities can cope with many changes in environmental conditions.

Of particular concern on this list are software faults and malicious attacks. Software dependability remains problematic and production software continues to be a weak link in computer systems. Similarly, malicious attacks are a concern because attacks against critical information systems have occurred and are expected to increase [8].

The need for increased survivability that stimulated this research project derives largely from concerns about software faults and security attacks, and it is these fault types that are the focus of the remainder of this paper.

2.1 Software Faults

The software that implements critical information systems is large, complex, and difficult to build. Many such systems involve tens of millions of lines of source text, and experience defect rates that are typical of large-scale software development [13, 15].

A common source of failure in information systems is the transition to operational status of a new version of a system. System upgrade, as this is called frequently, often leads to unanticipated difficulty even though the new version of the system might have been tested or otherwise checked. Making matters worse in many cases is that withdrawal of the new system and reversion to the old is difficult. This is because data formats and file contents have changed and production files contain updates resulting from temporary operation with the new system.

In practice, much of the effort expended during software development is in the form of fault avoidance and fault elimination, i.e., techniques that try to prevent software faults from being present in deployed systems. Despite careful attention to software fault avoidance and removal, production software tends to be the least dependable component in computer systems.

Various techniques have been developed that are designed to permit software to tolerate residual faults at execution time [5, 17]. These general approaches are derived from similar techniques that have been applied in hardware design to cope with hardware degradation faults. Such techniques attempt to detect and in some cases mask any error that arises in a software system thereby producing a general increase in dependability.

These methods have not proven very successful largely because the assumptions that hold under the use of fault tolerance in hardware do not generally hold for software.

The situation with large software systems remains therefore one in which high levels of dependability for critical systems, including critical information systems, is very hard to achieve and even harder to demonstrate.

2.2 Malicious Attacks

Although weaknesses have been demonstrated in some security techniques (encryption, protocols, mobile code such as Java[1], etc.), current security technology is quite strong in many areas. Despite this, information security has proved difficult to achieve in large modern information systems. Many problems have been reported in which supposedly secure systems have been penetrated and in some cases significant damage done. In other cases, denial of service have been brought about without penetration just by abusing the normal user interface.

In practice, it appears that many (perhaps even the majority) of serious security failures are attributable to software engineering defects in the systems experiencing the failure. Many systems have been penetrated, for example, by abusing the size limitations of a buffer to which an external agent has access. Such a penetration, referred to as a *buffer-overrun attack*, is possible because of an assumption that is not valid. The assumption is that no write would be attempted outside the storage allocated for the buffer, and it is made by the individual responsible for the implementation of the buffer. Sometimes the assumption is made implicitly.

It is inappropriate to refer to a successful buffer-overrun attack as a failure of security technology, although it is reasonable to think of it as a failure of security. It is not hard to check memory accesses during buffer accesses and ensure thereby that information outside the buffer is unaffected by buffer operations. The issue here is not security technology—the issue is the provision of systems that do not bypass otherwise effective security technology by extremely naive mistakes that provide access for those with malicious intent.

That many security failures are attributable to software engineering defects should not come as a surprise. In a large system, to avoid unintentional defects that can be exploited to defeat security requires (to a first approximation) that such defects be absent from large parts of the system. It is difficult to pinpoint the parts of a software system that might be exploited by an attacker, and so the problem we face is to build either:

- a system known to be free of defects throughout, or
- a system known to be free of defects in selected areas together with a rigorous argument (or proof) that the selected areas are the only ones in the system that might be vulnerable to attack.

This is a significant problem even for systems that might be described as “simple”. Modern information systems are not simple in any sense, however, because they are typically very large, often distributed, must maintain high availability, frequently run in real time or close to it, and so on.

1. See, for example, McGraw and Felten [12]

In summary, we conclude that software defects are important in their own right and they are the root cause of many security issues in information systems. A significant improvement in survivability, therefore, requires approaches that can help reduce the effects of software defects.

3 INFORMATION SYSTEMS CHARACTERISTICS

In this section we discuss several aspects of critical information systems that influence the approaches to software architecture that can be pursued in order to improve survivability. We begin by examining the system architecture that is at the heart of many of these systems. We then discuss the role of legacy software and COTS components.

3.1 System Architecture

From the users’ perspective, a critical information system is a single application despite the fact that the application is implemented on a large distributed system. In banking, for example, a large network exists to provide (among other things) a set of financial services for retail customers. The retail customer (i.e., the user) sees an interface that permits checks to be deposited, funds to be moved or withdrawn, and so on. To provide these services, each customer must be able to refer to any other customer’s account no matter where it is located within the system, i.e., within a large number of banks, so that funds transfers between customers can take place.

The various nodes in an information system cooperate in order to achieve the desired service goal, and different nodes maintain the data they need to perform their role. Cooperation usually involves nodes processing transactions such that the combination of transactions completes the service. In current implementations, it is very often the case that a small number (sometimes just *one*) of prescribed nodes have to carry out some of the transactions needed for certain user service requests. In terms of survivability, this immediately determines a single point of failure in the system.

Again, using retail banking as an example (and ignoring the many other services banks provide), there are thousands of leaf nodes in the network that provide customer access. They are connected in various ways to a much smaller number of intermediate nodes that provide regional service or centralized service associated with a specific commercial bank. Finally, those nodes are connected to a few nodes that provide communication for value transfer between separate commercial entities.

To illustrate application-domain characteristics in more depth, we use the actions required in clearing a check as an example. Check clearing is a complicated procedure that can be performed in a number of ways [7, 9]. We present a brief summary of *only one* of the ways in which checks are cleared—there are several variations.

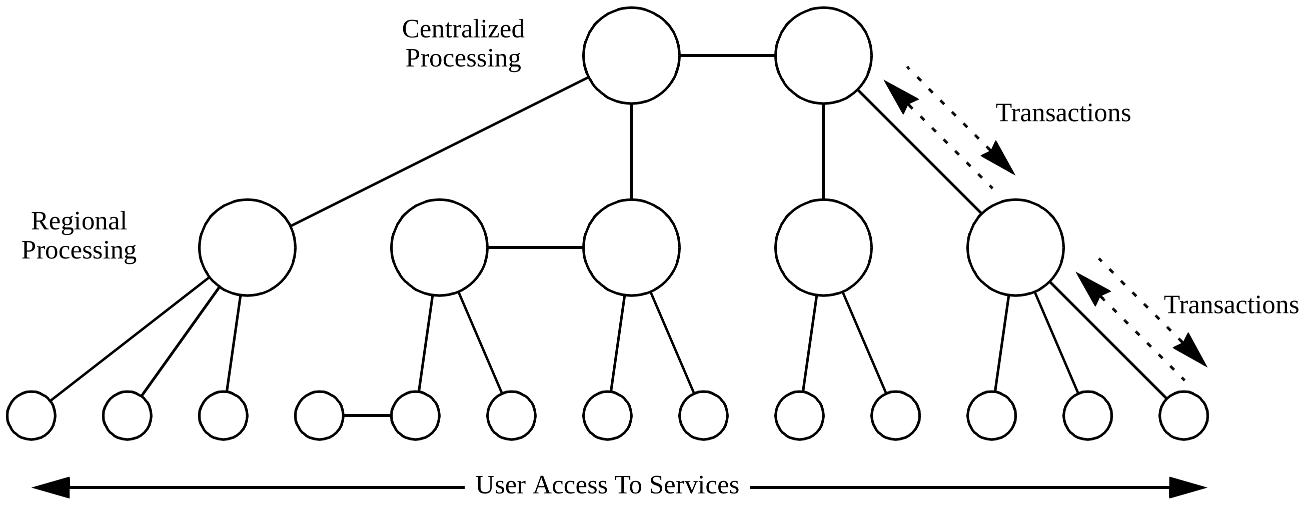

Fig. 1 - Hierarchic structure of critical information systems.

The recipient of a check presents the check for collection at a branch of the bank holding his or her account (i.e., a network leaf node). The check is scanned, the essential detail recorded (amount, payee, etc.) in electronic form, and then the paper check is often destroyed. The electronic form of the check is then forwarded to a regional processing center (i.e., a network intermediate node) where details are recorded in a database. The check is then forwarded to a clearing house or the Federal Reserve (i.e., a root node) where accumulation data describing what one retail bank owes or is owed by other retail banks is computed. Periodically this data is used to move funds between master accounts that the retail banks maintain. Finally, transactions flow back down through the system finally authorizing the recording of the deposit to one customer's account and a debit from another (although the funds will already have moved).

The general organization of many critical information systems is shown in Fig. 1. Note that this figure is hypothetical and shows *far* fewer nodes than are likely to be present in deployed systems. The connectivity of the network ends up having a hierarchic structure that is dictated by the communications needs of the transactions that are combined to provide the requisite service. It is important to note that alternative structures could be used but the hierarchic structure is the current form established in many domains (including retail banking services).

The hierarchic structure leads to critical vulnerabilities in information systems. The loss of a root node will stop service completely in a network where there is just one and reduce capacity severely in a network with more than one. Similarly, a common defect in the implementation of the leaf nodes (which are, to all intents and purposes, equivalent) will stop service. Finally, such vulnerabilities offer clear opportunity for malicious attack.

In summary, critical information systems tend to have the following characteristics:

- The provision of application-level service, i.e., as seen by the user of the service, depends upon the correct operation of several nodes and several network links.
- The degree of replication of nodes implementing the various application sub-function varies widely within the application.
- Different network nodes maintain different databases. These databases are essential to system operation.

3.2 Legacy Software

Legacy software is an essential part of most critical information systems. As these systems have been developed, existing software is often included as subsystems, and the size of the systems is such that there is a strong financial incentive to proceed in this way.

The inclusion of legacy components in critical information systems raises serious concerns about whether they will detract unacceptably from the required dependability. Legacy software is likely to have been in use for a long time and might have a satisfactory operational record in its current environment. But it is also likely that such software has a marginal (at best) maintenance record and might be the subject of concern about the success of future enhancements. Legacy software might not have certain basic functionality that is considered essential in critical information systems (encryption, data replication, and automated recovery, for example).

3.3 Commercial-Off-The-Shelf Components

Just as legacy systems are used in critical information systems, so are commercial-off-the-shelf (COTS) components, and for the same reasons. COTS components cannot necessarily be expected to address the reliability, availability, safety, or security needs of information systems (they almost certainly will not). The volume pricing and extensive functionality available with COTS software systems makes them very attractive in many instances, but their unproven dependability performance frequently deters and sometimes prohibits their use.

COTS systems are unknown quantities. It is unlikely that the manufacturer will make the source code for the system available to a customer, and, even if the source code were available, it would be very difficult to make necessary dependability assessments for such components if they are to be included in critical information systems. There is an important trade-off here. COTS provide tremendous functionality at low cost but very little essential development and assessment data usually comes along with the components.

4 APPLICATION COMPLEXITY

In practice, any significant property of a large software system is difficult to obtain. Reliability, availability, and security for example, all require that defects which defeat the provision of the property not be present. Availability is quickly compromised, for example, if a loss of service occurs because of a software defect and the system state cannot be re-established quickly. This will occur, for example, if a software defect leads to the corruption of data that is then propagated system-wide.

The situation with software contrasts with the provision of availability in the face of hardware failures attributable to degradation faults. Degradation faults have characteristics that permit useful assumptions to be made about the state of the system after failure and the failure interface that the device presents. This is why hardware replication is effective.

In order to try to establish significant properties of a large software system, it is tempting to turn to better software-engineering techniques. It might be argued, for example, that legacy and COTS components should not be used in critical information systems, and that, where such systems are to be deployed, they should be developed “from scratch”.

We claim that “from-scratch” systems would still not meet the dependability requirements because rebuilding would not yield systems that are sufficiently better than those we build now. Even if we were to rewrite from scratch the information systems upon which we depend (an economic impossibility), the result would be systems with significant numbers of faults, and these faults would detract from all areas of dependability. That we cannot engineer systems from scratch without defects is demonstrated every day.

The problem is that current and planned critical information systems contain so much software that, with the present state of the art, there is almost certainly no prospect of building software systems that can meet the dependability requirements that are beginning to appear. This is true no matter whether they are built from scratch, built with COTS components, built with legacy components, or any combination. The observed behavior of large software systems in the field shows dependability that is modest at best.

Thus, it is not merely COTS or legacy systems that fail to achieve or demonstrate dependability. Irrespective of the origin of the components, the complexity of large information systems is such that achieving high levels of dependability in any area varies from extremely difficult to essentially impossible. It is quite unrealistic to expect that a multi-million-line software system, no matter how carefully built, will be able to achieve the requisite levels of dependability if the complete system or a significant fraction of it has to be analyzed.

5 SHELLS

Given that current (or foreseeable) software technology offers no hope of assuring the dependability of the needed information system as a whole, the approach we are pursuing is to contain the problem in such a way that the necessary analysis can be performed.

What this means is that we have to develop systems in which only essential properties have to be established and in which those properties do not depend on the entire system for their demonstration. More specifically, it is necessary to:

- *limit* the dependability requirements, and
- *limit* the amount of software that has to be analyzed.

If these limitations diminish complexity to a point where an implementation with demonstrated dependability becomes feasible, there is some hope of building a survivable information system.

The dependability requirements have to be limited to those that are absolutely crucial. In a critical information system, for example, it is not possible to ensure that the system provides correct functionality. But it might be possible to ensure that certain minimal functions are performed even if faults in the system manifest themselves. Similarly, it might be possible to ensure that a system does no harm when a fault is manifested even though it fails to provide service. We note the analogy with fail-stop processors [18].

The amount of software that has to be analyzed can be limited by *localization*. For example, the localization of security concerns into a firewall [6] though imperfect in many senses is a strategy in which analysis can be largely localized.

Shells are the generalization of this localization idea. A shell in the sense used here is a layer of software that logically surrounds[1] a software artifact and either enforces some useful predicate on the state of system of which the artifact is a component or supplements the functionality of the artifact in some crucial way. We refer to this predicate as a *policy* [11]. The artifacts that would be surrounded in the distributed systems of interest here would be the software running on the nodes, i.e., the software on each node would be surrounded by its own shell structure.

Shells require information in order to be able to achieve their goals. Thus, typically they will include mechanisms to intercept and examine application communication.

Critically, however, shells permit the localization of dependability enforcement to a point where properties of interest in a number of areas are concentrated into a relatively small part of the software. Thus, provided the localization mechanism can be shown to be effective (a considerable issue, we admit), only the shells themselves need significant verification efforts.

In part, the role of a shell can be thought of as protecting an application element from a dangerous world (e.g., protection of an application from a security threat) and protecting a vulnerable world from a dangerous application (e.g., protection of the remainder of a system and its context from an application element that has been penetrated by an attacker).

A limited version of the shell notion has been demonstrated with security kernels and guards [2] and, more recently, with enforcement safety kernels [4, 10, 16]. Enforcement safety kernels illustrate the shell concept since they are designed to provide assurance that certain safety policies are enforced irrespective of the actions of the remainder of the software system. Thus, defects in what amounts to a very large software system cannot compromise the safety of the system since the safety kernel will enforce needed safety policies. The analysis process could thus be limited to the safety kernel to a large extent.

An important notion introduced with enforcement safety kernels was that of the *weakened policy* [10]. Many policies have to be stated in terms that related to the application in such a way that implementation of policy enforcement software becomes unwieldy and this defeats the object of the kernel. The solution is to use a weakened policy in which an extension has to be made to the application so that it does the bulk of the computation associated with the policy. The weakened policy implemented by the kernel is then a check that the application does the policy computation correctly.[2]

A technology known as “wrappers” has been pursued in a number of contexts and wrappers share some commonalities with shells. Wrappers however, have been developed primarily to supplement deficient existing applications in some specific way or to permit integration of otherwise incompatible components. Thus, for example, an existing application that lacks encryption on a critical data link might be “wrapped’ with the encryption being implement in the wrapper in a manner that is transparent to the application.

The critical conceptual difference between wrappers and shells is that shells are used because they provide the localization property rather than because they permit a software artifact to be enhanced without change. In other words, we might choose a shell architecture for a system from the outset and build the system around this notion. Thus, a shell architecture is quite appropriate as the basic structure of a new system being developed. In that case the artifacts being surrounded by a shell are being developed with the knowledge that a shell will be present. In addition, an existing system might be enhanced to include shells in order to achieve certain dependability goals.

5.1 Types of Shell

There are three types of properties that shells can implement and this leads to three types of shell—protection, enhancement, and correction:

- *Protection Shell.*
  The notion of a protection shell derives from earlier work on security and safety kernels. In the overall survivability architecture, the role of a protection shell is to enforce a predicate irrespective of what the remainder of the software does.
- *Enhancement Shell.*
  An enhancement shell provides essential enhancement functions to an application. In particular, it implements the application supplements necessitated by weakened policies, although the policies themselves are implemented by the protection shell.
- *Correction Shell.*
  A correction shell’s role is to correct known but uncorrected or suspected defects in an application. An example of a known defect is a system that is susceptible to buffer-overrun attacks but where the buffer-management software has not been corrected.

The logical structure of an application supplemented with all three types of shell is shown in Fig. 2. All three types of shell have access to application communication but each provides the specific service for which it was designed.

It is important that the implementation of the three shells itself be survivable. In other words, the implementation structure should facilitate demonstrating that the shell implementations cannot be stopped by external software defects or security attacks. We deal with these issues in section 6.

1. Hence the origin of the term shell.
2. This is a necessarily brief and incomplete review of these ideas. For a more thorough discussion see Knight and Wika [10].

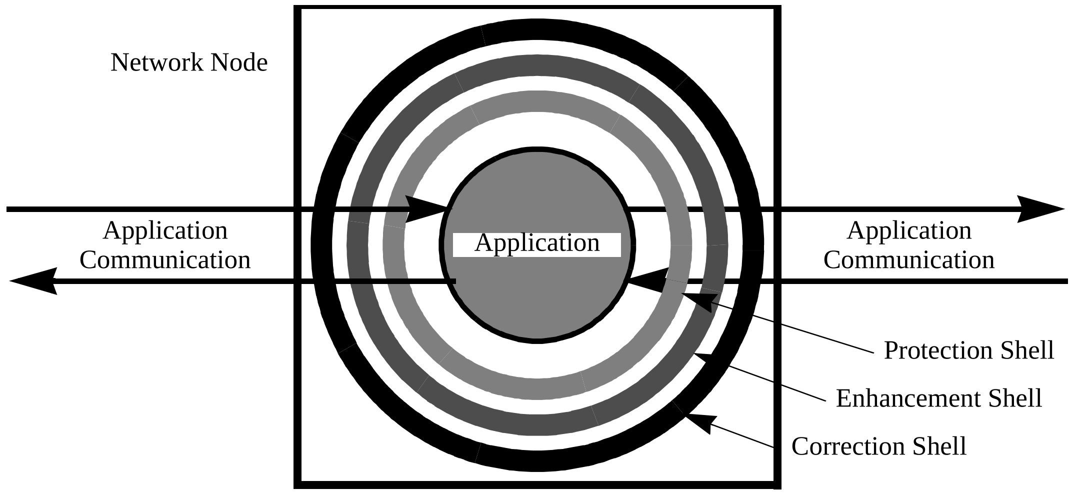

Fig. 2 - The logical organization of the three shell types and application software for a single network node.

5.2 Transparent and Opaque Shells

Most proposals for the use of wrappers assume that their presence is transparent to the artifact being wrapped. In other words, the artifact sees its operating environment as unchanged and the artifact does not have to be modified in order to permit it to be wrapped.

An important question that arises in the implementation of wrappers is how to achieve this style of implementation. This is an important question for shells also because they will be used with COTS components for which modification is probably not possible, and with legacy systems for which modification is not desirable. We refer to an implementation in which no application software is modified as *transparent* and the implementation as *opaque* otherwise.

From the perspective of shell implementation, transparency has significant advantages and disadvantages. The advantages include:

- The application does not have to be changed and so systems are less affected by the introduction of a shell.
- Details of the application, such as the source code are not needed.
- The majority of the shell implementation is application independent so only one is needed for each operating system as opposed to one for each application.

These are important advantages, but the disadvantages are significant also:

- The shell implementation has to operate with no assistance from any source. Thus, for example, to gain access to the activities of an application, a shell has to intercept some fraction (possibly a large fraction) of the application communication without disturbing the application. This leads to very difficult implementation strategies in which, for example, all system calls have to be intercepted.
- The shell has to try to reconstruct what the application was doing from the information that it intercepts. This might involve rebuilding application data structures (including files) and other state information. This tends to make the shell unwieldy.
- A transparent implementation implies that shells operate by examining only the information flowing between the artifact that they surround and the environment in which that artifact operates. In practice, even if this information can be captured, it is often at such a low semantic level that the shell cannot determine what the artifact was in fact doing. Worse even than this is the fact that much of the information that the shell needs to operate might not even pass between the artifact and the environment.
- The transparent approach is one in which a certain amount of programming gymnastics is required. This does not seem like good engineering practice, and we suspect that it might not be a source of confidence for engineers building critical information systems. They are likely to be more confident in a system with a comprehensive engineering solution.

We conclude that true transparency is usually impractical and requiring it could be an unnecessary restriction on an otherwise useful idea. If a transparent implementation is used in circumstances that are inappropriate, the result is likely to be poor engineering that will likely lead to reduced rather than improved dependability.

In the implementation that we are developing, transparency is used where practical. In practice, however, we hypothesize that the engineers building critical information systems will be quite amenable to making small changes to application systems if they permit significant services to be obtained from a shell architecture.

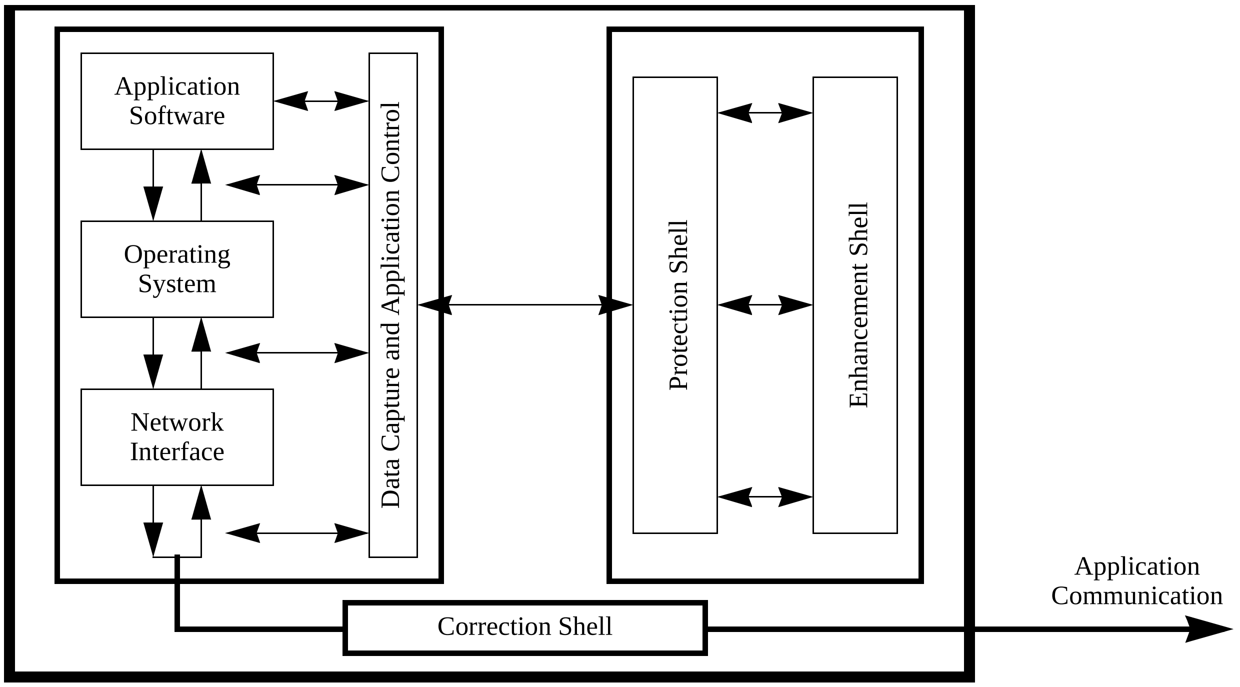

Fig. 3 - Node implementation structure for survivability architecture.

6 SHELL IMPLEMENTATION

The implementation of shells in general is far more difficult that it appears from the simple statement of the idea. There are two main areas of implementation difficulty. The first is that any artifact that one wishes to surround with a shell is almost certainly tightly integrated with many components of an existing system. These components, the operating system, perhaps a database system, and so on, have complex interfaces, and the shell has to interact with the artifact of interest and other components. This is a non-trivial undertaking at best.

The second area of implementation difficulty is in shell protection. The shells must continue to operate when all around them might be failing. In addition, the shells must be protected against malicious attack so that they do not represent a new vulnerability that can be exploited.

We can only describe briefly the implementation strategy that we are developing for nodes in the architectural approach to survivability. It is summarized in Fig. 3. The significant aspects of this implementation structure are the following:

- The shells operate on different computers from those that execute the node's application element. This is to ensure continued operation when the application fails and to limit the means of communication between the application and the shells. Were arbitrary communication possible (such as through memory accesses) a shell could be disrupted by erroneous software or by an intruder.

  An alternative implementation in which the shells operate merely as separate processes might be considered if the associated protection mechanisms could be shown to be sufficiently strong.
- The correction shell operates in series with the application communication. It is by this means that the correction shell ensures that all known and suspected deficiencies in the application that might threaten the security of the node are corrected.
- The protection shell receives the data that it needs for policy enforcement from a "data capture" component that is added to the application software. This component intercepts application communication to the extent necessary but it is also the primary interface used by the modifications introduced into the application to communicate with the protection shell.

  Since these modifications are present to gather information about application actions at a higher semantic level than is possible with external communications, they are mostly limited to intercepting and thereby recording function calls in the application. This is a minor perturbation for the most part.

An important aspect of the shell approach that we are developing is the use of direct synthesis of the protection and correction shell implementations from a set of policy specifications. This technique has been demonstrated and evaluated with the enforcement safety kernel implementation and found to be highly satisfactory. The synthesis approach provides the following three benefits:

- It permits the development of application-specific shells very rapidly.
- It allows for a shell to be revised very quickly (even in the field) if the dependability requirements change or if the initial set are found to be deficient.
- It permits a variety of different shells to be developed rapidly for the different needs that will arise inevitably in distributed applications.

The first and third of these benefits ensure the practical viability of the approach. The second benefit allows the approach to be applied by “trial and error” when, as frequently occurs, the allowable behaviors are not known a priori. This second benefit also allows newly discovered defects in the application that threaten the survivability goals to be quickly dealt with by installing new policies in the correction shells.

7 SURVIVABILITY ARCHITECTURES

Strengthening the nodes in a critical information system using shells does not guarantee system-wide survivability properties. Indeed, the very design of the shells as described so far limits their attention to the node that they are surrounding.

With shells in place, however, their existence can be exploited to produce a comprehensive design that permits significant system-wide dependability properties to be established. These properties are established under important but fairly reasonable assumptions by analyzing the shells as individual entities together with the overall design. We refer to the resulting system as a *survivability architecture*. Since the shells combine to enforce predicates on the system state, their combined action is to permit the enforcement of a predicate (perhaps consisting of several conjuncts) on the entire system state.

The exploitation of the shells to establish system-wide dependability properties requires the implementation of an additional mechanism—system-wide shell-to-shell communication. Only if shells can operate cooperatively to achieve some goal can system-wide properties be established.

Consider, as an example, an information system in which one of the nodes is a single-point of failure, i.e., it undertakes to perform at least one transaction for every user service request that is offered. The loss of this node is clearly critical yet its loss in a network of possibly thousands of other nodes will usually be manifested by a delay that will look like a denial of service.

A far better strategy is to permit the remainder of the network to undertake some alternative service algorithm while the critical node is unavailable. In the banking example used earlier, if a critical node was lost and funds transfers could not be achieved as a result, a strategy in which all remaining network nodes were appraised of the situation and were able either to queue user service requests in an orderly manner or provide some form of reduced service would be a reasonable approach to survivability.

A second example can be seen in the provision of security in critical information systems. Despite the use of a variety of authentication mechanisms, penetrations do occur. To deal with them, intrusion detection mechanisms are often employed that raise an alarm if the signature of an intruder is detected.

Despite (but not ignoring) the obvious opportunity for abuse that an inter-shell communication mechanism provides, the possibility exists if one is present for a network-wide response to intrusion if a single node detects an intrusion. Appraising the entire network of the situation would allow more extreme security measures to be deployed quickly, the signature of the intruder to be distributed, and so on.

8 CONCLUSION

There are many critical information systems that provide services to society which are crucial, and keeping these systems operating satisfactorily is a significant challenge. Most present systems provide good service but as the number of such systems and society’s dependence on them increases, the requirement for information system survivability will increase correspondingly.

The types of failure to which an information system is subject are many but most can be dealt with by existing means. Those that cannot tend to be either software or security defects. However, observation suggests that the majority of security incidents are in fact attributable to defective implementations, i.e., software defects, rather than failures of security technology. We claim, therefore, that software defects are the primary problem which have to be dealt with in increasing survivability.

Critical information systems have a hierarchic structure and this together with the fact that such systems incorporate both COTS and legacy elements severely restricts the solution approaches that can be pursued.

We conclude that an approach to survivability based on system software architecture is a viable approach. We are developing a survivability architecture for critical information systems based on shells—layers of software that surround an application and provide a variety of services. These services include the enforcement of important application policies, extension of the application to implement essential supplementary functions, and the implementation of corrections to known or suspected application defects.

Each node in the system is enhanced with shells that permit various useful properties to be established about the node. However, a key feature of the survivability architecture is that the shells can communicate, and this mechanism is used to permit system-wide error detection, recovery, and continued service. This permits a much faster reaction to problems that threaten the network and ensures that optimal survivability strategies can be implemented where they exist.

An implementation of the survivability architecture based on nodes using the Microsoft Windows NT operating system is presently being developed.

ACKNOWLEDGMENTS
It is a pleasure to acknowledge the assistance we have received in understanding the details of the financial infrastructure from Bob Trent and Patrick Marx. This work was funded in part by DARPA through Department of the Air Force grant number F30602-96-1-0314 and in part by NASA under grant number NAG-1-1123-FDP.

REFERENCES

1. President's Commission on Critical Infrastructure Protection, Available at <http://www.pccip.gov/>
2. S. Ames, M. Gasser and R. Schell, Security kernel design and implementation: An introduction, *IEEE Computer*, 16: 14-22, (July 1983).
3. R.H. Anderson, A.C. Hearn, The day after in cyberspace II, Rand Corporation Report Number MR-797-DARPA, The Rand Corporation, Santa Monica, CA, March 1996.
4. A. Burns and A. Wellings, Safety kernels: specification and implementation, *Journal of High Integrity Systems*: 1(3), 1995.
5. L. Chen and A. Avizienis, N-version programming: A fault-tolerance approach to reliability of software operation, in *Digest of Papers FTCS-8: Eighth Annual International Conference on Fault Tolerant Computing*, pages 3-9, Toulouse, France, June 1978.
6. W. R. Cheswick, S. M. Bellovin, *Firewalls and Internet Security: Repelling the Wily Hacker*, (Addison-Wesley Professional Computing)
7. R. Clair, J. Kolson, K. Robinson, The Texas Banking Crisis and the Payments System, *Federal Reserve Bank of Dallas: Economic Review* 1st Qtr 1995, page 14.
8. Department of Defense, Report Of The Defense Science Board Task Force On Information Warfare - Defense (IW-D) November 1996 Office Of The Under Secretary Of Defense For Acquisition & Technology Washington, D.C. 20301-3140, Available at <http://www.jya.com/iwd.htm>.
9. B. Geva, FedWire Transfer of Funds, *Banking Law Journal*, 104(5), 423-424, Sep/Oct 1987.
10. J.C. Knight and K.G. Wika, On the Enforcement of Software Safety Policies, *Proc. COMPASS: Conference on Computer Assurance*, Gaithersburg, MD, June 1995.
11. D.A. Marriott, Management Policy Specification, Technical Report DoC 94-1, Department of Computing, Imperial College of Science Technology and Medicine, November 1993, Available at <ftp://dse.doc.ic.ac.uk/dse-papers/management/policy_spec.ps.Z>
12. G. McGraw and E.W. Felten, *Java Security*, John Wiley and Son, 1997.
13. M. Neil and N. Fenton, Predicting Software Quality using Bayesian Belief Networks, In *Proc. of 21st Annual Software Engineering Workshop*, pages 217-230, NASA/Goddard Space Flight Center, Greenbelt, MD, December 1996.
14. P. Neumann, *Computer Related Risks*, Addison Wesley, New York, 1995.
15. S. Pfleeger and L. Hatton, Investigating the Influence of Formal Methods, *IEEE Computer*, 30(2): 33-43, (February 1997).
16. J. Rushby, Kernels for Safety?, in *Safe and Secure Computing Systems*, pages 210-220, T. Anderson Ed., Blackwell Scientific Publications, 1989.
17. B. Randell, System Structure for Software Fault Tolerance, *IEEE Transactions on Software Engineering*, SE-1 (2), pages 220-232 (June 1975).
18. R.D. Schlichting and F. B. Schneider, Fail-Stop Processors: An Approach To Designing Fault-Tolerant Computing Systems, *ACM Transactions On Computer Systems*, 1, pages 222-238, August 1983.