Survivability through Customization and Adaptability: The Cactus Approach

Matti A. Hiltunen, Richard D. Schlichting, Carlos A. Ugarte, and Gary T. Wong
Department of Computer Science
The University of Arizona
Tucson, AZ 85721, USA
E-mail: hiltunen/rick/cau/gary@cs.arizona.edu.

Abstract

Survivability, the ability of a system to tolerate intentional attacks or accidental failures or errors, is becoming increasingly important with the extended use of computer systems in society. While techniques such as cryptographic methods, intrusion detection, and traditional fault tolerance are currently being used to improve the survivability of such systems, new approaches are needed to help reach the levels that will be required in the near future. This paper proposes the use of fine-grain customization and dynamic adaptation as key enabling technologies in a new approach designed to achieve this goal. Customization not only supports software diversity, but also allows customized tradeoffs to be made between different QoS attributes including performance, security, reliability, and survivability. Dynamic adaptation allows survivable services to change their behavior at runtime as a reaction to anticipated or detected intrusions or failures. The Cactus system provides support for both fine-grain customization and dynamic adaptation, thereby offering a potential solution for building survivable software in networked systems.

1. Introduction

A survivable system is one that is able to complete its mission in a timely manner, even if significant portions are incapacitated by attacks or accidents [3]. Systems connected to large networks such as the Internet are faced with additional challenges—they must prevent unauthorized use, maintain confidentiality, and provide adequate service to proper users [19]. While survivability builds on research in security, reliability, fault-tolerance, safety, and availability, it is more than just a combination of these—it is also about the interaction of these different properties [61]. A survivable system must be able to continue providing service despite attacks or accidents. To do so, it must have facilities to protect against such threats, detect threats when they occur, and react to counteract threats and recover from damage. Thus, building a survivable system architecture requires not just exploiting techniques from security, fault-tolerance, safety, and real time, but it also requires developing new techniques.

We contend that *fine-grain customization* and *dynamic adaptation* are both key enabling technologies for survivability in networked systems. Fine-grain customization allows a service behavior to be tailored in fine detail to the requirements of the service user and the execution environment. Our approach to customization is based on *configurability*, which is the ability to easily construct customized service variants out of a set of software modules. Dynamic adaptation is the ability to change the behavior of a service by modifying the service configuration quickly at runtime. Such adaptation may involve, among other things, loading new code modules dynamically. A system based around these technologies can support customized survivability solutions, as well as evolve these solutions in both the short and long term to react to threats or to take advantage of new techniques.

Specific advantages of using fine-grain customization and dynamic adaptation in this context include the following:

- Services can be configured from existing collections of modules realizing different techniques for protection, detection, and reaction, rather than built from scratch.
- Configuration choices can be made depending on the perceived threats at a given time, with explicit control over tradeoffs such as the cost of an intrusion detection technique versus its coverage.
- Different survivability strategies for peer services can be adopted depending on the system scale, e.g., a closely-coupled strategy for collections of servers on

a LAN versus a highly decentralized strategy for a corporate intranet.

- Multiple protection and detection techniques can be used in combination within a single service to further enhance survivability, e.g., successive authentication layers to protect sensitive data.
- Multiple customized services can be used in combination to protect sensitive aspects of system operations.
- Artificial diversity can be achieved easily by making different choices for a service configuration on different machines.
- Systems and individual services can adapt dynamically to respond to changing circumstances, such as a detected threat or a heightened state of alert (e.g., a wartime situation).
- New survivability techniques can be introduced dynamically into services as developed.

Done properly, a survivable system based on customization and dynamic adaptation could be a unifying framework that leverages both existing and future survivability techniques.

The Cactus Project at the University of Arizona is developing an infrastructure to support the fine-grain customization and dynamic adaptation capabilities needed to construct this type of survivable system. This paper introduces the Cactus model for configurable and dynamically adaptive distributed services, describes Cactus services relevant to survivability, and outlines potential future applications of Cactus in the survivability context.

2. The Cactus approach

The Cactus system is a framework for constructing highly customizable and dynamically adaptable middleware services for networked systems. Cactus provides an integrated approach that addresses a wide range of functional properties, as well as quality of service (QoS) attributes such as reliability, timeliness, performance, and security. The fine-grain customization provided by the systems allows application-specific control over these attributes and their potential tradeoffs.

2.1. The Cactus model

The Cactus model provides tools for building highly configurable distributed protocols and services. In Cactus, different properties and functions of a service are implemented as software modules that interact using an event-driven execution paradigm. A service in Cactus is implemented as a *composite protocol*, with each semantic variant of a QoS attribute or other functional component within the composite protocol implemented as a *micro-protocol*. A micro-protocol is, in turn, structured as a collection of *event handlers*, which are procedure-like segments of code that are executed when a specified *event* occurs. Events are used to signify state changes of interest, such as “message arrival from the network”. When such an event occurs, all event handlers bound to that event are executed. Events can be raised explicitly by micro-protocols or implicitly by the Cactus runtime system. Event handling operations are implemented by the Cactus runtime system that is linked with the micro-protocols to form a composite protocol. The use of events provides indirection between micro-protocols that facilitates configurability and adaptability.

The model provides a number of operations for controlling events and handler execution. The two most important are:

- *bid* = **bind**(*event, handler, order, static_args*): Specifies that *handler* is to be executed when *event* occurs. *order* is a numeric value specifying the relative order in which *handler* should be executed relative to other handlers bound to the same event. When the handler is executed, the arguments *static_args* are passed as part of the handler arguments.
- **raise**(*event, dynamic_args, mode, delay*): Causes *event* to be raised after *delay* time units. If *delay* is 0, the event is raised immediately. The occurrence of an event causes handlers bound to the event to be executed with *dynamic_args* (and *static_args* passed in the **bind** operation) as arguments. Execution can either block the invoker until the handlers have completed execution (*mode* = SYNC) or allow the caller to continue (*mode* = ASYNC).

Other operations are available for unbinding handlers from events, creating and deleting events, halting event execution, and canceling a delayed event. Execution of handlers is atomic with respect to concurrency, i.e., each handler is executed to completion before the execution of the next handler is started. The binding of handlers to events can be changed at runtime.

In addition to a flexible event mechanism, Cactus supports two other features useful for configurable and adaptive service: service variables and the Cactus message abstraction. Service variables are variables and data structures shared by all micro-protocols within a given composite protocol. Such variables can be used to store those portions of the service state that need to be accessed by multiple micro-protocols. For example, a communication service would typically have a data structure storing messages that have been transmitted, but cannot be discarded yet. The atomic execution of event handlers makes it easy to use the service variables since it eliminates most concurrency problems.

The Cactus message abstraction generalizes the typical message format consisting of a message body and a header. In particular, a Cactus message consists of a body and a dynamic set of named attributes that have scopes corresponding to the composite protocol (*local*), the protocols on a single machine (*stack*), and the peer protocols at the sender and receiver (*peer*). The local attributes correspond to local variables associated with the message and the peer attributes correspond to normal header fields in a message. The stack attributes allow controlled information sharing across protocol layers when necessary.

Micro-protocols can add, read, and delete message attributes using operations `setAttr()`, `getAttr()`, and `deleteAttr()`. Two micro-protocols can share an attribute if they agree on the name of the attribute. A `pack()` operation adds any peer attributes to the message body, while an analogous `unpack()` operation strips those attributes from the body. The default pack operation includes in the message body all the necessary information for unpacking, including attribute names and lengths. However, the pack and unpack operations are completely customizable so if the message format is known *a priori* by the sender and receiver, this additional information can be omitted. The custom packing and unpacking also makes it possible to create standard message formats in case a composite protocol is interacting with a peer not implemented using Cactus. Finally, this facility makes it possible to use other header processing techniques such as header compression transparently to the micro-protocols that use the attributes.

The Cactus message abstraction makes it easy to manage message headers for highly configurable services. Typically, each combination of micro-protocols would require its own header format that includes those fields needed by the chosen micro-protocols. The message abstraction eliminates the need to explicitly construct the appropriate header format since micro-protocols can simply add the needed attributes to a message. This feature also makes it easy for the message format to change under different operating modes and to piggyback information on messages. In particular, a micro-protocol that wants to piggyback information can do so by adding a peer attribute with an appropriate attribute name at the sender. At the receiver, the corresponding micro-protocol can check if the information is piggybacked on a message simply by invoking `getAttr()` with the same attribute name as an argument. If the attribute is not piggybacked on this particular message, `getAttr()` returns an error indication. This flexibility also allows a message to carry only those attributes—i.e., header fields—that are required for that message.

The Cactus model promotes a *two-level composition* approach to constructing subsystems: a service is constructed out of micro-protocols and a subsystem is constructed out of services. This approach is illustrated in Figure 1. This figure depicts a communication subsystem consisting of composite and traditional protocols. In the middle is a composite protocol, which contains shared service variables, including a bag of messages, micro-protocols, and some events. The boxes in the middle represent micro-protocols, while to the right are events with binding between handlers and events represented by arrows.

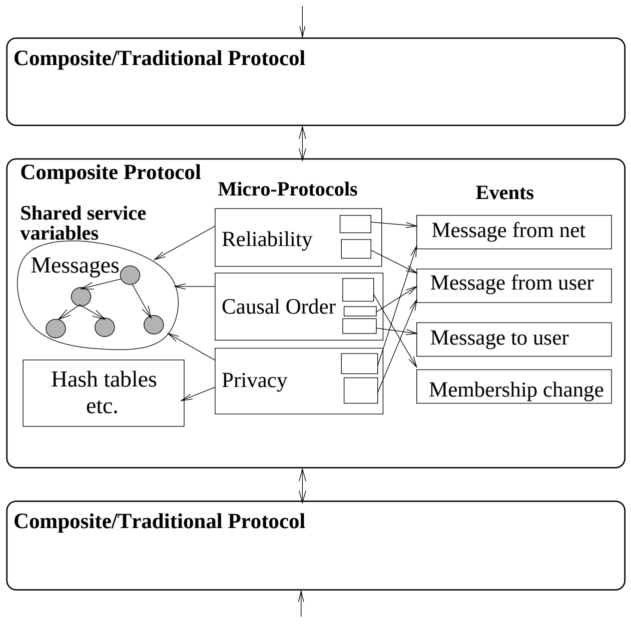

Figure 1. Composite protocol.

The Cactus model does not dictate the external interface provided by a service implemented as a composite protocol. For example, the service may export a CORBA object interface with methods. In this example, the service is a communication protocol that resides in the protocol stack on top of a lower level communication service such as UDP. In such a case, interaction between protocols is either via a standard Uniform Protocol Interface (UPI) such as that supported by the *x*-kernel [35] or via a separate systemic interface that supports vertical integration of support for Quality of Service (QoS) requests and modifications.

Four major prototype implementations of Cactus are currently available or under development. One implementation runs on OpenGroup/RI Mach MK 7.3 using C and CORDS, a variant of the *x*-kernel [59]. A separate version of this implementation called CactusRT supports construction of configurable service with real-time constraints [34]. Another version called Cactus++ is a C++ based implementation that runs on Linux and Solaris. Finally, a Java implementation of the model called Cactus/J runs on numerous operating systems, including Windows NT, Linux, and Solaris. All released Cactus prototypes and services are available through the Cactus home page http://www.cs.arizona.edu/cactus/.

2.2. Configurability in Cactus

All the mechanisms provided by the Cactus model are designed to support configurability by promoting independence between micro-protocols. For example, a micro-protocol may bind an event handler to an event without knowing which micro-protocol will raise the event, or raise an event without knowing which micro-protocol, if any, has handlers bound to this event. Similarly, the message abstraction allows micro-protocols to add new message attributes to a message without coordination with other micro-protocols. As a result, micro-protocols typically do not need to know about one another, which facilitates construction of customized variants of a service by configuring together the desired set of micro-protocols.

Our design approach to constructing highly-customizable services is illustrated in Figure 2. First, the useful abstract properties or execution guarantees of a service are identified by exploring existing implementations of the service. Second, the useful abstract properties are implemented as configurable modules. Note that there may be multiple alternative implementations of a property, where each implementation is optimized for certain execution environment. The properties of a service, as well as the modules that implement the properties, have fundamental relations that dictate which combinations are feasible. The goal of the implementation is not to introduce any additional restrictions on configurability beyond those found in these relationships [29].

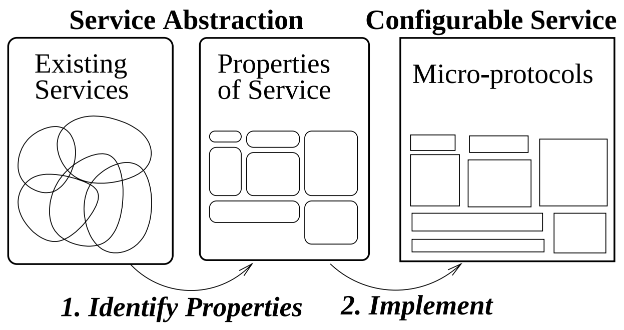

Figure 2. Design of a configurable service

Cactus services with survivability features are discussed below in section 4, but a number of other configurable services have also been implemented. A group membership service provides a range a customization options related to event ordering, synchrony, partition handling, and failure detection [33]. A group remote procedure call service allows customization of message ordering, call synchrony, atomicity, and reply collation [8]. Finally, a real-time communication service allows customization of message ordering, reliability, message deadlines, and the number and roles of communication participants [34]. Other services under development include a distributed shared memory service and a distributed file service.

2.3. Adaptability in Cactus

In addition to fine-grain configurability, the Cactus model offers features for supporting adaptive solutions. The indirection provided by the Cactus mechanisms also makes it easier to construct adaptive services that change their behavior as a result of changes in the execution environment or in the service requirements. In particular, the event mechanism allows new micro-protocols to be activated by simply binding their event handlers to the appropriate events or old micro-protocols to be deactivated by unbinding their handlers. Moreover, service variables make it easy to transfer state from old to new micro-protocols, while the message abstraction makes it trivial to modify the message header format due to an adaptive change from one algorithm to another.

The Cactus model supports different methods of adaptation, ranging from changing the execution parameters of a micro-protocol to actually switching the set of micro-protocols used to implement a service. Two approaches are available for changing the set of micro-protocols: event-handler rebinding and dynamic code loading. In the case of rebinding, the composite protocol is compiled with the alternate behaviors implemented as different micro-protocols, and a mode switch only involves unbinding the old event handlers of the deactivated micro-protocols and binding the handlers of the activated micro-protocols. In the case of dynamic code loading, new micro-protocols are loaded into a running composite protocol. After the new code is loaded, it can be activated in the same manner as above. The Cactus model makes such runtime code modification easier and allows more coordination than simply loading procedures into an ordinary running program because of the decoupling between micro-protocols provided by the event-driven execution model. We have experimented with dynamic code loading using dynamic libraries on Solaris, as well as using Java facilities for class loading in Cactus/J.

The ability to change the behavior of a running program is only the first step in adaptation. In particular, in the case of networked systems, the more difficult problem is often the coordination of the adaptation at the different sites involved in the computation. To coordinate this process, we have developed a three phase model for adaptations consisting of change detection, agreement, and action [32]. The agreement phase, where the different sites decide if the adaptation is necessary and what the adaptation should be, can be relatively complicated. To simplify the implementation of this phase, we have developed a library of consensus micro-protocols. The current set of consensus protocols are fault-tolerant, but do not currently address other aspects of survivability. However, new consensus micro-protocols could easily be added with other features such as authentication or tolerance to Byzantine failures.

We are in the process of defining a Cactus-based software architecture for constructing highly-adaptive configurable services [11]. In this architecture, a configurable service consists of adaptive and static (non-adaptive) components, where each component implements a QoS attribute, property, or function of the service. The service components interact using events and service variables. Each adaptive component is implemented as a collection of micro-protocols, including a number of adaptation-aware micro-protocols (AAMPs) and a component adaptor micro-protocol that coordinates adaptations between the AAMPs (Figure 3). Each AAMP provides a different implementation of the component functionality, with the component adaptor switching between the alternative AAMPs depending on the state of the execution environment and the service requirements. For example, a component that provides communication security may switch between different cryptographic micro-protocols depending on the level of security provided by the underlying network. Since the service components do not interact directly—e.g., by invoking methods on one another—the AAMPs can be switched without any effect on other system components.

The AAMPs are somewhat different than normal micro-protocols. In particular, they export an adaptation interface that consists of operations for evaluating the fitness of the micro-protocol for the current execution environment, and operations for activating and deactivating the micro-protocol. Adding this interface does not require any changes in the Cactus model, however, since an adaptation aware micro-protocol can simply be defined as a subclass of an ordinary micro-protocol with the new adaptation operations.

The software architecture for adaptive systems has two major design goals: to minimize the overhead of adaptability during normal operation and to minimize the impact of the adaptive process itself on the application using the adaptive service. The event mechanism provides a natural solution to the first goal. Since the new AAMP can simply bind its event handlers to the appropriate events, no additional redirection is required. Thus, the only overhead during normal operation is the system monitoring needed to detect changes in the execution environment. The second goal is achieved by allowing the adaptation to occur in phases, where both the old and new AAMP are partially active during certain phases. The Cactus model facilitates this approach, since any number of event handlers may be bound

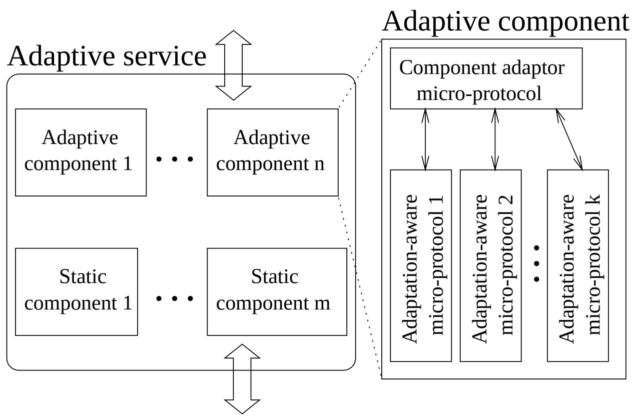
Adaptive service
Adaptive component 1
Adaptive component n
Static component 1
Static component m
Adaptive component
Component adaptor micro-protocol
Adaptation-aware micro-protocol 1
Adaptation-aware micro-protocol 2
Adaptation-aware micro-protocol k

Figure 3. The structure of an adaptive service.

to an event at the same time. Furthermore, the consensus protocols execute in the background without halting normal processing. As a result, an application using an adaptive service should experience at most a slight temporary slowdown in service during adaptation.

3. Survivability through customization and adaptability

The fine-grain QoS customization and dynamic adaptability supported by Cactus provide powerful mechanisms for survivability. This section outlines how mechanisms supported by Cactus can be utilized in the survivability context.

3.1. Fault tolerance and security

Fault tolerance and security, two methods for increasing survivability [39], are two of the QoS attributes provided by many customizable Cactus services. Different fault-tolerant micro-protocols can be developed to tolerate different types of failures. Omission failures of the underlying communication network can be tolerated by using different types of retransmission micro-protocols. Examples of such micro-protocols include traditional positive and negative acknowledgment based protocols, multiple transmission protocols, and atomicity protocols for group communication where the group members collectively ensure that if at least one member receives a message, all members will eventually receive it. Message corruption failures can be tolerated using either traditional message checksums or cryptographic checksums. The latter have the ability to detect intentional message modifications. Processor failures can be tolerated by using micro-protocols that implement object or process replication (active or passive) or message logging. Alternative micro-protocols can also be developed to tolerate different failure models ranging from crash to arbitrary failures [30].

Similar to fault tolerance, different types of security micro-protocols can be provided to deal with various situations. For communication security, security micro-protocols can provide privacy, authenticity, and message integrity using cryptographic techniques. Micro-protocols can also protect from replay attacks by adding sequence numbers, timestamps, or other unique nonces to messages. Even non-repudiation can be provided by having a micro-protocol log messages. Similar techniques can also be applied to realize other types of data security. For example, files in a file system can be encrypted for privacy or protected from tampering by using cryptographic checksums.

Customization allows each user or system administrator to determine how much security and fault tolerance is required at any given time. An increase in fault tolerance or security requires more resources, and increases the response time and reduces the throughput of a service. Thus, it is important that the levels of fault tolerance and security are set based on the perceived threats at any given time, as well as the requirements of the current applications and users. For example, the security requirements for an email service used by the military can differ based on whether the country is currently involved in a conflict, what information is being transmitted, and who is sending and receiving the information.

Two different approaches can be used for providing fault tolerance and security guarantees using Cactus: integrating the code implementing the enhanced QoS with an existing service or providing it as a separate fault tolerance or security service. In the integrated approach, an existing Cactus service such as RPC or group communication is enhanced to provide additional fault tolerance or security by developing new micro-protocols. For example, fault tolerance through replication can be added to the RPC service as a micro-protocol that sends each request to a collection of servers instead of one. Similarly, security can be added to any communication service by adding encryption, authentication, or message integrity micro-protocols.

The second alternative implements fault tolerance and security guarantees as a separate service. An example of such a fault-tolerance service is the group membership described in [33], while an example of a security service is the secure communication service described in [31]. Such specialized services provide customizable fault-tolerance or security abstractions for higher level services and applications.

The choice between these alternatives is case specific. The integrated approach allows fault tolerance and security micro-protocols to be totally customized for the particular service. For example, the security requirements may be different for different message types used by the service. However, since the integrated approach increases the number of micro-protocols, it makes the service more complicated. The flexibility of the two-level composition model in Cactus allows the use of the design alternative that best fits the requirements.

3.2. Diversity

Diversity methods attempt to make it difficult for an intruder to exploit known security vulnerabilities by making different instances of a program or an operating system different enough that one method of breaking the service might not work on others [13, 14]. Diversity methods include wrapper programs that filter inputs to a potentially buggy program [60], inserting additional checks into programs, and permutations of program code and data layout.

The configurability supported by Cactus provides a natural mechanism for constructing diverse programs and services. Different configurations of Cactus services have some degree of natural diversity given that users can customize the service to their exact requirements and execution environment. Additional diversity can be provided by implementing alternative micro-protocols for different service properties and QoS attributes. For example, communication privacy can be provided using numerous different encryption methods (e.g., DES, RSA, IDEA, Blowfish) or combinations of encryption methods. If the intruder does not know which method or combination of methods is used for the encryption, it becomes more difficult to break the encryption.

Similar diversity techniques can be applied to other security services. For example, a configurable intrusion detection service [16] could be constructed using Cactus, where new intrusion detection methods are added as new micro-protocols and various combinations could be used together. Similarly, the authentication service of an operating system could be customized for each OS installation.

3.3. Adaptability

Adaptability has numerous current and future applications in survivability, including automated reaction to intrusions. For example, methods for stopping an intrusion after it has been detected by terminating a suspected connection or by preventing the intrusion from spreading to other computers [9] can be viewed as adaptive behavior. Automatic actions to restore the system state after an intrusion and to prevent future exploitation of the same security holes can be considered another type of adaptation. Traditional recovery methods used for fault tolerance can often be utilized for recovery of this type as well, although the additional problem of identifying data and programs that may have been modified must also be solved. Moreover, most adaptations related to intrusion tolerance are currently not even automated. For example, responses to CERT advisories must be implemented manually, leading to delays and non-uniform deployment.

Adaptability can also be used as a preventive mechanism by allowing a service to change its behavior dynamically based on user requirements and perceived or detected threats. For example, if an intrusion detection service suspects an intrusion, other services can adaptively increase their security levels to prevent the intruder from gaining more information or doing further damage. Such preventive mechanisms must naturally be combined with corrective actions if damage was done before detection.

We are currently exploring adaptability related to a number of QoS attributes in the context of Cactus. Some of this work focuses on performance adaptations, that is, changing algorithms within a service to increase performance given the current execution environment [11]. We have also explored using adaptability to implement fault tolerance, ranging from considering failure repair as an adaptive action [32] to considering algorithms that adapt to a change in the failure models exhibited by the underlying system [10]. Future work will include adaptive security to accommodate changed user requirements and detected or perceived threats, and adaptive real time to accommodate changes in available resources and system workload.

Both adaptive security and adaptive real time are relevant for survivability. The use of adaptation to increase security when an intrusion is detected is obvious, but other types of security adaptations can also be used to enhance survivability. For example, if an intrusion reduces available resources, a service may adaptively weaken its security guarantees to reduce CPU resource requirements. The weakened security can then be compensated for by frequently switching different low cost security mechanisms. Note that this type of adaptation is an example of a trade off between two different QoS attributes, security and performance.

Adaptive security introduces a number of challenges that still need to be solved. For example, the consensus algorithms used to determine if an adaptation is required and which adaptation should be made must be made intrusion tolerant using message authentication or Byzantine methods. It may also be important to disguise the adaptations from the intruder to hide the original detection. Current research is addressing these issues.

Adaptive real time is also important for survivability, since a survivable system is often required to provide a timely response in spite of failures and intrusions [3]. If failures or intrusions reduce the available set of resources, adaptive services must either reallocate resources or reduce the resource usage of some of its constituent parts. The adaptive functionality provided by Cactus supports this process in several ways. One is by facilitating construction of services that can provide a reduced level of service when resources are reduced. Another is by making it easy to effect tradeoffs between different resources, e.g., by changing to a compression algorithm that requires more CPU time in order to reduce the network bandwidth required. Of course, a primary challenge in this area is to design and implement adaptive actions that can execute in bounded time in order to meet the application's real-time constraints.

3.4. Enhancing survivability transparently

It is often necessary to increase the survivability of existing legacy or off-the-shelf applications. This can be accomplished either by replacing underlying communication and operating system services with survivable versions, or by transparently inserting new middleware services between the application and the underlying services.

We have experimented with the base functionality need to realize the first approach for communication services on both MK 7.3 and Linux. On MK 7.3 with CORDS, all or part of a protocol stack to be inserted into the kernel, which can be used to replace existing communication services provided by the operating system. On Linux, kernel loadable modules can be used to achieve similar functionality. Currently we are experimenting with the use of loadable modules on Linux to simulate the characteristics of a wireless network on top of a wired Ethernet, but similar techniques could be used to insert survivable behavior into the kernel.

There are a number of options for replacing other operating system services with Cactus equivalents depending on the OS platform. One option is to modify the operating system kernel directly if the source code is available (e.g., Linux). Other options include instrumented connectors in NT [2] and, again, loadable kernel modules. While kernel modifications are difficult for the intruder to circumvent, the other options may not provide 100 % protection from intruder circumvention.

We are also exploring the transparent insertion of new middleware services. In the context of CORBA, our goal is to enhance the QoS for existing CORBA clients and servers without modifying either their code or the underlying ORB. We are working on accomplishing this goal on the Orbix ORB by inserting composite protocols into smart proxies at clients and filters at servers, and on the Visibroker ORB using smart stubs at clients and wrappers at servers. We are also working on inserting new middleware services by intercepting signals on Linux and Solaris. This approach is being used in a distributed shared memory service in which the shared memory abstraction is implemented by intercepting segmentation fault signals related to access attempts.

4. Survivable Cactus Services

A number of configurable services built using Cactus have important survivability features. This section discusses some existing services, and their current and planned survivability features.

4.1. Secure communication service

SecComm, a secure communication service implemented using Cactus, allows fine-grain customization of a range of security attributes including privacy, authenticity, message integrity, replay prevention, and non-repudiation [31]. A secure connection created through SecComm is customizable in the sense that only the required security attributes are guaranteed. Furthermore, the strength of guarantee associated with each attribute can be customized at a fine grain level. For example, the service offers the choice of the cryptographic techniques and key lengths used to implement each attribute, and each attribute can be guaranteed using arbitrary combinations of security algorithms. Using combinations of algorithms can make it harder for intruders to break system security because they do not know the method that needs to be broken, although methods based on diversity and secret methods such as this are not uniformly accepted by the security community. Nonetheless, SecComm allows each user to choose between using proven methods such as DES or RSA, using secret combinations of secret methods, or using combinations of proven and secret methods to further increase security.

SecComm exemplifies and extends a current trend in network security design that allows customization of security attributes for explicitly managing the cost/benefit tradeoff. For example, IPSec [37], a collection of protocols being developed by the IETF to support secure packet exchange at the IP layer, provides two security options. The *authentication header* (AH) option does not encrypt the data contents of the packet, but provides optional authenticity, integrity, and replay prevention by adding an AH that contains a cryptographic message digest. The *encapsulating security payload* (ESP) option provides privacy by encrypting the data contents of the packet and optional authenticity, integrity, and replay prevention using a message digest. Naturally, due to the enhanced support for configurability provided by Cactus, SecComm provides more flexibility and extensibility than existing secure communication services.

Figure 4 illustrates the main micro-protocol classes and their interactions through events in SecComm. Note that each component in this figure represents a class of micro-protocols, e.g., the privacy class includes DESPrivacy, RSAPrivacy, and OneTimePadPrivacy micro-protocols.

In the current version, a custom secure communication session is created by selecting the desired set of

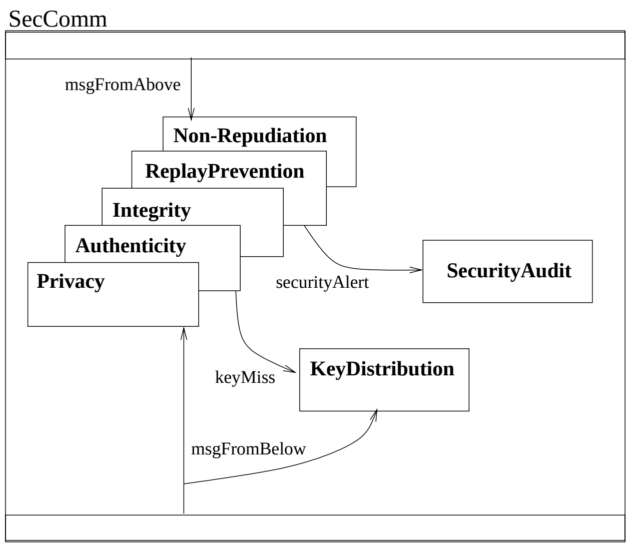

Figure 4. Micro-protocol classes and their interactions.

micro-protocols. Future versions could include higher-level configuration interfaces in which the user specifies only the required level of assurance for each property. The micro-protocol combination that satisfies these requirements would then be created automatically using information on, for instance, relative strengths of different encryption methods.

Future work related to SecComm will focus on adding adaptability features to the protocol. This type of adaptive security service must be able to increase the level of security when requested by the user or when an intrusion attempt is detected, and it must be able to reduce the resource usage of the service when performance is not satisfactory. An interesting challenge will be determining if adaptation can be inexpensive enough that reasonably high level of security can be provided by switching frequently between different inexpensive cryptographic methods. A special instance of this type of protocol switch would be altering the cryptographic key at runtime.

4.2. GroupRPC service

The GroupRPC service allows a client to execute a procedure call on one or more servers. Such facility can be useful for a number of reasons, but the main survivability aspect of such facility is that it can provide fault tolerance through replication. Unfortunately, providing fault tolerance of this type can be expensive. One important factor determining this cost is the type and number of server failures to be tolerated. Typically, the type of failures to be tolerated is expressed in terms of *failure models* that range from relatively benign crash or omission failures to arbitrary (or Byzantine [41]) failures. In terms of the amount of redundancy and execution time required, tolerating a processor crash is much cheaper than tolerating Byzantine failures. Furthermore, the cost is determined not only by the failure model, but also by the number or frequency of the failures expected to occur. Any realistic system can in principle exhibit failures in any of the failure models, but typically, failures in the more benign classes are more frequent than severe failures. Thus, the choice of failure model for a particular task should be based on the frequency of different types of failures in the given execution environment, as well as the criticality of the task. The tradeoff, of course, is that making stronger assumptions about failures improves the performance of the system, but lessens the degree of fault coverage provided by the system and thus the reliability of the task [48].

We have implemented a configurable GroupRPC service that allows the failure model and number of failures to be tolerated to be customized depending on the needs of the application [30]. The service includes micro-protocols for each failure model that implement the appropriate detection and membership algorithms. For example, if the service is configured to tolerate Byzantine failures, the servers run Byzantine agreement protocols to agree on the set of requests to be processed, as well to agree as any membership changes of the server group. Thus, any faulty or compromised server or client cannot cause the server group to become inconsistent. In addition to the algorithms used, the choice of failure model dictates how many failures can be tolerated, that is, how many server replicas are required. Thus, in order to tolerate M server failures, we need M+1 replicas for crash, send omission, and late timing failures, 2M+1 replicas for receive omission and early timing failures, and 2M+1 replicas for Byzantine failures assuming message authentication [30].

Table 1 illustrates how the change in the failure model affects the response time [30]. Each of the configurations listed tolerates one failure of the given type for configurations that guarantee atomicity and total ordering for the calls. The large increase of response time in the Byzantine case is due to slow RSA-based message authentication and an agreement algorithm that operates in synchronous rounds. This concrete example illustrates the need for customization between reliability and performance, and thus, survivability and performance.

Failure Model	Clients	Servers	Time
None	1	1	3.6
	2	1	5.9
Crash	1	2	6.2
	2	2	13.4
Send omission and late timing	1	2	6.6
	2	2	13.8
Receive omission and early timing	1	3	10.5
Byzantine	1	3	18924

Table 1. Average response time (in ms)

4.3. System monitoring services

Two distributed system monitoring services have been constructed using Cactus. One, implemented using Cactus++, runs on Solaris, while the other, implemented using Cactus/J, runs on numerous operating systems including Linux and Windows NT. Both services are structured as one centralized global monitor that provides the user a GUI for monitoring and controlling the system and a number of local monitors executing on each of the machines being monitored. In the current versions, the local monitors are user processes that use standard operating system interfaces to collect information about the system load, CPU and memory usage of processes, current users, etc. Each aspect of the system behavior is monitored and reported by a separate micro-protocol, which makes it easy to configure a service that monitors only the necessary information. This structure also makes the service easily extensible since new micro-protocols can be added to monitor other aspects of the system behavior. The Cactus++ version is extensible at configuration time, i.e., the monitoring service must be recompiled and relinked to include any new micro-protocols. The Cactus/J version, on the other hand, is adaptive, i.e., new micro-protocols can be added dynamically at runtime. This functionality is implemented using a specialized micro-protocol class loader.

These monitoring services can easily be extended to provide monitoring oriented towards security and survivability. For example, the tools can be used to detect denial of service attacks by constructing micro-protocols that collect resource utilization information and produce warnings when thresholds are exceeded. A resource being fully utilized—such as CPU, memory or network buffers—might indicate that a denial of service attack is underway.

Monitoring micro-protocols can also be used to oversee other security-critical aspects of system operation. For example, one can monitor the password file and report any changes. Such a distributed monitoring architecture can also make it more difficult for intruders to cover their tracks by modifying log files, since the monitor can record any changes in the log files and store them on the global monitor. Although a considerable amount of useful information can be collected by just monitoring the file system and using operating system calls, more powerful survivability monitoring would require instrumenting existing code in the operating system as well as services such as ftp, telnet, and http.

Naturally, changes are also needed in the monitoring service itself to make it survivable. For example, it must be impossible for an intruder to modify or replay messages sent by a local monitor or to replace a genuine local monitor by one that does not report any suspicious activity. Message modification and replay can easily be prevented using authentication and message integrity methods similar to those used in SecComm, or the monitoring service could just be built on top of SecComm. Local monitor replacement can be prevented, or at least made very difficult, by having the global monitor give the local monitor a private authentication key at startup. Similarly, the intruder must not be able to impersonate the global monitor to the local monitors. This would be particularly dangerous in the Cactus/J version since it allows new micro-protocols to be loaded at runtime. The natural solution is again authentication of all messages sent by the global monitor. Note that the configurability provided by Cactus makes it easy for each installation to customize the type and strength of authentication used.

5. Future Survivability Services

Cactus could be used as a framework to implement a number of other survivability services with customizability and adaptability features. In the following, we discuss some potential survivability services and their implementation using Cactus.

5.1. Intrusion detection services

Research on intrusion detection has traditionally focused on auditing system logs and more recently, on automated intrusion detection [16]. Intrusion detection systems analyze audits provided by the computing environment to detect attacks or misbehavior occurring in the environment. Two major approaches can be identified. The *misuse-detection approach* detects known signatures or symptoms of attacks, while the *anomaly-detection approach* detects deviations from an assumed normal behavior of the system. Misuse-detection is generally more accurate, but can only account for known attacks. Tools such as statistics [43], expert systems [21], signature analysis [40], neural networks [15], genetic algorithms, and model based reasoning [26] have been used to implement detection of either approach. Detection may focus on analyzing information from one computer (host-based), e.g., [28, 62], inspecting packets passed through a network (network-based), e.g., [49, 53], or some combination, e.g., [46]. Recently, research on intrusion detection has focused on integrating existing intrusion detection systems by providing a Common Intrusion Detection Framework (CIDF) that provides protocols for interaction and collaboration [57].

The customization abilities provided by the Cactus framework could be used as the basis for implementing a highly-configurable intrusion detection service (IDS). The micro-protocol approach would make the service easily extensible in the sense that new micro-protocols could be added to collect information from new sources (e.g., log files, operating system, other system components), to analyze information, and to react to any suspected intrusions. The new analysis micro-protocols could include ones developed for newly recognized misuse patterns. The analysis could also use a combination of different methods (e.g., signature analysis, neural networks, model based reasoning) and use voting or agreement to determine if an intrusion should be suspected. The reaction micro-protocols could range from ones that notify an operator to others that automatically attempt to stop the intrusion.

A configurable IDS designed in this way has a number of other advantages, such as allowing tradeoffs to be customized. For example, it becomes possible to control the tradeoff between the coverage of the detection and its performance and resource utilization. It also allows customization of detection coverage versus the number of false detections, and thus, the potential inconvenience caused to legitimate users. A configurable IDS could also be customized depending on the perceived threat and the current mode of operation. For example, the system could be designed to react automatically to suspected intrusions that occur during nighttime hours when system operators are not at work and few real users are using the system. During daytime operation, the service could be more conservative in its actions and pass responsibility for most reactions to the system operators.

The detected or suspected intrusions detected by an IDS are fundamental for initiating adaptation in other survivability services, but an IDS could also adapt itself. Quick adaptations could occur at the time a threat is suspected, for example, to add additional detection mechanisms focused on the suspected activities. Furthermore, an IDS could adapt in the long term in the sense of evolving to incorporate countermeasures to newly detected attack methods. In particular, newly discovered intrusion signatures might be added to detection micro-protocols, and new detection micro-protocols could be introduced into the IDS at runtime. The changes between different modes of operation could also occur adaptively, rather than through reconfiguration. For example, an IDS could adapt by increasing its sensitivity when system administrators suspect that an intrusion might occur.

The primary goal of developing a Cactus-based IDS would not be to develop new intrusion detection methods, but rather to leverage existing and future methods into a configurable adaptive framework. That is, individual intrusion detection methods could be implemented as micro-protocols, which would allow different variants of the service to be created by configuring chosen subsets of the methods. Any coordination required between the different detection methods, such as weighted voting, could easily be implemented as a separate coordinator micro-protocol that is designed to handle any detection configuration.

5.2. Survivable data storage services

Survivability can be very important for data storage services such as file systems and database systems. A survivable data storage service (SDSS) must not only ensure confidentiality and integrity of data, but also provide data availability in spite of, and after, failures and intrusions. Techniques such as data fragmentation and replication can be used to achieve both confidentiality and availability even if some of the computers are successfully compromised [17, 22, 38]. If the data cannot be physically protected—that is, only authorized access is allowed in all cases—confidentiality and integrity can be implemented using cryptographic techniques similar to those used for secure communication. Furthermore, other techniques such as checkpointing and logging of changes can be applied to allow restoration of data after an intrusion. Finally, a part of the information may be stored outside of the SDSS to enhance survivability. For example, decryption keys or file checksums may be stored on a smartcard that the user provides to the system on demand. Note that a SDSS can use an existing underlying data storage service such as a file system for actually storing the data. Thus, in the case of a distributed file system such as NFS, the actual file server does not need to be modified.

Configurability can be applied to an SDSS as easily as it can be applied to the secure communication service—the different options for service guarantees can be implemented as separate micro-protocols. Note that one SDSS may store data with different survivability requirements. This means that the service must be configured with a set of micro-protocols that is sufficient to satisfy all the guarantees for all the data currently in the system. For each data item accessed, only the required micro-protocols will be activated. The information of which micro-protocols are used for each data item could be stored transparently in the SDSS itself, or this information can be provided by the user when they try to access a data item.

Adaptability can be used for SDSSs in the form of corrective actions if an intrusion and data destruction or modification is detected. Changing the survivability methods on the fly is more difficult for storage services since it could be expensive, for example, to decrypt all files and then encrypt them again using a new method.

5.3. Access control and authentication services

An authentication service (AS) is used to ensure the identity of a principal (i.e., a user or a program operating on a user's behalf), and an access control service (ACS) is used to determine for each file and other system resource if a principal is allowed to use it in some specified manner. Each operating system implements its own authentication (e.g., password, one time password, smartcard, and physical characteristics) and access control methods (e.g., access control lists, capabilities, and Unix-style permission bits).

The authentication and access control methods provided by operating systems are relatively rigid and do not provide much support for survivability. In particular, intruders that manages to pass the authentication test (e.g., guess root password) are free to do any amount of damage. Thus, the higher levels of customization provided by Cactus would be a useful feature for survivable systems.

Allowing such services to be customizable would open a range of possible solutions with better survivability characteristics than traditional systems. For example, authentication could be customized to be one time only at login time, periodic at fixed time intervals, or usage based depending on what files and other resources the user attempts to access and use. Such combinations of access control and authentication could have many interesting applications. For example, accessing some important system files might require that the user is executing as root and that they know an additional password required to modify the file. Such additional security boundaries could also be specified for directories, resulting in arbitrary sequences of successive security boundaries for accessing important system resources. Another approach would be to specify a level of authenticity required for accessing each system resource and executing additional authentication steps when a user attempt to access something higher than the current level. The authenticity level of a user could also reduce over time to at least partially handle the problem of someone else using a session left open by a legitimate user. Since authentication and access control are typically services provided by the operating system, it is not easy to replace them using Cactus equivalents. Some options described in section 3.4 could be applied here depending on the platform.

Survivability adaptations useful for an access control service include revoking all access rights from a compromised user identity and increasing authentication requirements on system resources if the system is attacked. Adaptations in the case of an authentication service include reauthenticating users that have used authentication challenges that have been compromised, as well as replacing authentication methods that have been compromised. Furthermore, the overall required authentication level of the whole system may be increased, in which case all current users are reauthenticated. The signal to perform for such adaptations would typically come from other system components such as the intrusion detection service.

6. Related work

6.1. Configurability

Configurability or extensibility has gained increased attention during the last decade. In the area of communication protocols, notable examples are *x*-kernel [35], Adaptive [56], Horus [51], and Ensemble [27]. The *x*-kernel presents a model for hierarchical composition of modules with identical uniform protocol interfaces (UPIs). The Horus and Ensemble projects adopt the same hierarchical model, but extend the UPI to include a larger set of operations. Adaptive uses a service specific back-end with slots for different functions of the service, where each function can be implemented using a range of alternative modules. Extensibility has also been used in operating systems, e.g. SPIN [6], Scout [45], and Exokernel [20], and database systems, e.g. RAID [7] and Genesis [4].

Customizability is beginning to gain greater attention in the area of communication security. A number of Internet standards under development include some customization features, including IPSec [37], Secure Socket Layer (SSL) [25], Transport Level Security (TLS) [18], Secure HyperText Transfer Protocol (S-HTTP) [52], and Privacy-Enhanced Mail (PEM) [42]. These protocols typically allow a choice of privacy and message integrity methods, as well as optional authentication methods. A number of research projects have also addressed customizable security. For example, the *x*-kernel model has been used to construct configurable protocol stacks for secure communication [47]. Recently, TIS Labs has worked on composable, replaceable security services [58], and the Ensemble project has added a choice of customizable cryptographic techniques for data communication [54].

Customizability is naturally very useful also in system monitoring services, and most services offer some way of modifying the service behaviors. For example, Pulsar [23] supports extensibility by allowing new Tcl scripts to be written for collecting additional information, satool [44] uses Unix scripts, and CARD [1] uses Perl scripts. Some monitoring tools, such as Rscan [55], have been designed to detect security loopholes. Many intrusion detection tools have similar type of extensibility features, including Emerald [46]. In general, however, the use of configurability in the survivability context has been limited and most examples have focused on secure communication and system monitoring, including intrusion detection.

In contrast with service-specific frameworks that allow some extensibility, Cactus is a general framework specifically designed for fine-grain configurability and extensibility. Thus, the possibilities for extensibility are much greater. For example, it would be easy to add security and reliability features into our system monitoring tool, whereas it would probably be difficult to add such features to many other monitoring tools where extensibility is limited.

6.2. Adaptability

Adaptability has been proven to be a useful feature in particular for networking protocols. For example, the Transmission Control Protocol (TCP) of the Internet protocol suite uses adaptive mechanisms for flow control, retransmission, and congestion control [36]. Other examples include adaptive media access control, e.g. [12], adaptive encoding and compression, e.g. [24], and adaptive routing algorithms, e.g. [5]. However, adaptability in such protocols is typically limited to changing some parameters that control the execution the protocol, e.g., timeout values or transmission windows, rather than switching the protocol. To our knowledge, the only other work addressing runtime algorithm switching is work done in Ensemble [27, 50]. In contrast with our work that allows each protocol or service component to adapt with minimal impact on other parts of the system, Ensemble requires that a new protocol stack be created with the new protocol. Message transmission must then be interrupted while the adaptation is in progress.

7. Conclusions

This paper has outlined potential uses of the Cactus approach for implementing various aspects of survivability. In particular, we claim that the fine-grain customizability and dynamic adaptability supported by Cactus can have a significant impact on the survivability of networked systems. Fine-grain customization allows each user or system administrator to make individual survivability versus performance tradeoffs, as well as provides a foundation for using artificial diversity for survivability. Adaptability allows services to react to attacks and failures when they do occur, and facilitates their evolution when new attacks or countermeasures become available.

References

[1] E. Anderson and D. Patterson. Extensible, scalable monitoring for clusters of computers. In *Proceedings of Eleventh Systems Administration Conference (LISA '97)*, pages 9–16, San Diego, CA, Oct 1997.

[2] R. Balzer. Instrumented connectors for mediating architecture interactions. DARPA Windows NT Workshop, Seattle, WA. http://www.dyncorp-is.com/darpa/meetings/-win98aug/Balzerinstruments.html, Aug 1998.

[3] M. Barbacci. Survivability in the age of vulnerable systems. *IEEE Computer*, 29(11):8, Nov 1996.

[4] D. Batory, J. Barnett, J. Garza, K. Smith, K. Tsukuda, B. Twichell, and T. Wise. GENESIS: An extensible database management system. *IEEE Transactions on Software Engineering*, SE-14(11):1711–1729, Nov 1988.

[5] P. Bell and K. Jabbour. Review of point-to-point network routing algorithms. *IEEE Communications Magazine*, 24(1):34–38, 1986.

[6] B. Bershad, S. Savage, P. Pardyak, E. Sirer, M. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety, and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain Resort, CO, Dec 1995.

[7] B. Bhargava, K. Friesen, A. Helal, and J. Riedl. Adaptability experiments in the RAID distributed database system. In *Proceedings of the 9th Symposium on Reliable Distributed Systems*, pages 76–85, 1990.

[8] N. Bhatti, M. Hiltunen, R. Schlichting, and W. Chiu. Coyote: A system for constructing fine-grain configurable communication services. *ACM Transactions on Computer Systems*, 16(4):321–366, Nov 1998.

[9] P. Brutch, T. Brutch, and U. Pooch. Electronic quarantine: An automated intruder response tool. In *Proceedings of the Information Survivability Workshop 1998*, pages 23–27, Orlando, FL, Oct 1998.

[10] I. Chang, M. Hiltunen, and R. Schlichting. Affordable fault tolerance through adaptation. In J. Rolin, editor, *Parallel and Distributed Processing, Lecture Notes in Computer Science 1388*, pages 585–603. Springer, Apr 1998.

[11] W.-K. Chen, M. Hiltunen, and R. Schlichting. Software architecture for building adaptive software. Technical report, Department of Computer Science, University of Arizona, Tucson, AZ, 1999. In preparation.

[12] M. Choi and C. Krishna. An adaptive algorithm to ensure differential service in a token-ring network. *IEEE Transactions on Computers*, C-39(1):19–33, 1990.

[13] C. Cowan and C. Pu. Immunix: Survivability through specialization. In *Proceedings of the 1997 Information Survivability Workshop*, Feb 1997.

[14] C. Cowan and C. Pu. Survivability from a Sow's ear: The retrofit security requirement. In *Proceedings of the Information Survivability Workshop 1998*, pages 43–47, Orlando, FL, Oct 1998.

[15] H. Debar, M. Becker, and D. Siboni. A neural network component for an intrusion detection system. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 1–11, Oakland, CA, 1992.

[16] D. Denning. An intrusion-detection model. *IEEE Transactions on Software Engineering*, SE-13(2):222–232, Feb 1987.

[17] Y. Deswarte, J.-C. Fabre, J.-M. Fray, D. Powell, and P.-G. Ranea. Saturne: A distributed computing system which tolerates faults and intrusions. In *Proceedings of the Workshop on Future Trends of Distributed Computing Systems*, pages 329–338, Hong Kong, Sep 1990.

[18] T. Dierks and C. Allen. The TLS protocol, version 1.0. Request for Comments (Standards Track) RFC 2246, Certicom, Jan 1999.

[19] R. Ellison, D. Fisher, R. Linger, H. Lipson, T. Longstaff, and N. Mead. Survivable network systems: An emerging discipline. Technical Report CMU/SEI-97-TR-013, Nov 1997.

[20] D. Engler, M. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 251–266, Copper Mountain Resort, CO, Dec 1995.

[21] M. Esmaili, R. Safavi-Naini, and J. Pieprzyk. Computer intrusion detection: A comparative survey. Technical Report Ref. 95-07, Center for Computer Security Research, University of Wollongong, Australia, May 1995.

[22] J.-C. Fabre, Y. Deswarte, and B. Randell. Designing secure and reliable applications using fragmentation-redundancy-scattering: an object-oriented approach. In *Proceedings of the 1st European Dependable Computing Conference*, pages 21–38, Berlin, Germany, Oct 1994.

[23] R. A. Finkel. Pulsar: An extensible tool for monitoring large Unix sites. *Software Practice and Experience*, 27(10):1163–1176, 1997.

[24] A. Fox, S. Gribble, E. Brewer, and E. Amir. Adapting to network and client variation via on-demand, dynamic distillation. In *Proceedings of the 7th ASPLOS Conference*, Oct 1996.

[25] A. Freier, P. Karlton, and P. Kocher. The SSL protocol, version 3.0. Internet-draft, Netscape Communications, Nov 1996.

[26] T. Garvey and T. Lunt. Model-based intrusion detection. In *Proceedings of the 14th National Computer Security Conference*, Washington, DC, Oct 1991.

[27] M. Hayden. The Ensemble system. Technical Report TR98-1662, Department of Computer Science, Cornell University, Jan 1998.

[28] L. Heberlein, K. Levitt, and B. Mukherjee. A method to detect intrusion activity in a networked environment. In *Proceedings of the 14th National Computer Security Conference*, pages 362–371, 1991.

[29] M. Hiltunen. Configuration management for highly-customizable software. *IEE Proceedings: Software*, 145(5):180–188, Oct 1998.

[30] M. Hiltunen, V. Immanuel, and R. Schlichting. Supporting customized failure models for distributed software. *Distributed Systems Engineering*, 1999. To appear.

[31] M. Hiltunen, S. Jaiprakash, and R. Schlichting. Exploiting fine-grain configurability for secure communication. Technical Report 99-08, Department of Computer Science, University of Arizona, Tucson, AZ, Apr 1999.

[32] M. Hiltunen and R. Schlichting. Adaptive distributed and fault-tolerant systems. *Computer Systems Science and Engineering*, 11(5):125–133, Sep 1996.

[33] M. Hiltunen and R. Schlichting. A configurable membership service. *IEEE Transactions on Computers*, 47(5):573–586, May 1998.

[34] M. Hiltunen, R. Schlichting, X. Han, M. Cardozo, and R. Das. Real-time dependable channels: Customizing QoS attributes for distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):600–612, Jun 1999.

[35] N. Hutchinson and L. Peterson. The *x*-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, Jan 1991.

[36] V. Jacobson. Congestion avoidance and control. In *Proceedings of the SIGCOMM '88 Symposium*, pages 314–332, Aug 1988.

[37] S. Kent and R. Atkinson. Security architecture for the internet protocol. Request for Comments (Standards Track) RFC 2401, BBN Corp, Home Network, Nov 1998.

[38] H. Kiliccote and P. Khosla. Borg: A scalable and secure distributed information system. In *Proceedings of the Information Survivability Workshop 1998*, pages 101–105, Orlando, FL, Oct 1998.

[39] P. Krupp, J. Maurer, and B. Thuraisingham. Survivability issues for evolvable real-time command and control systems. In *Proceedings of the 1997 Information Survivability Workshop*, Feb 1997.

[40] S. Kumar and E. Spafford. A pattern matching model for misuse intrusion detection. In *Proceedings of the 17th National Computer Security Conference*, pages 11–21, Oct 1994.

[41] L. Lamport, R. Shostak, and P. M. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, Jul 1982.

[42] J. Linn. Privacy enhancement for internet electronic mail: Part I: Message encryption and authentication procedures. Request for Comments RFC 1421, Feb 1993.

[43] T. Lunt, R. Jagannathan, D. Anderson, C. Dodd, F. Gilham, C. Jalali, H. Javitz, P. Neuman, A. Tamaru, and A. Valdez. System design document: Next-generation intrusion detection expert system (NIDES). Technical report, SRI International, Mar 1993.

[44] T. Miller, C. Stirlen, and E. Nemeth. satool — A system administrator's cockpit, an implementation. In *Proceedings of Seventh Systems Administration Conference (LISA '93)*, pages 119–129, Monterey, CA, Nov 1993.

[45] A. Montz, D. Mosberger, S. O'Malley, L. Peterson, T. Proebsting, and J. Hartman. Scout: a communications-oriented operating system. In *Proceedings of the 1st Symposium on Operating Design and Implementation (OSDI)*, page 200, Nov 1994.

[46] P. Neumann and P. Porras. Experience with EMERALD to date. In *Proceedings of the 1st USENIX Workshop on Intrusion Detection and Network Monitoring*, Santa Clara, CA, Apr 1999.

[47] H. Orman, S. O'Malley, R. Schroeppel, and D. Schwartz. Paving the road to network security or the value of small cobblestones. Technical Report 94-16, Department of Computer Science, University of Arizona, Tucson, AZ, May 1994.

[48] D. Powell. Failure mode assumptions and assumption coverage. In *Proceedings of the 22nd IEEE Symposium on Fault-Tolerant Computing*, pages 386–395, 1992.

[49] M. Puldy and M. Christensen. Lessons learned in the implementation of a multi-location network based real-time intrusion detection system. In *Proceedings of the 1st International Workshop on the Recent Advances on Intrusion Detection*, Louvain-la-Neuve, Belgium, Sep 1998.

[50] R. v. Renesse, K. Birman, M. Hayden, A. Vaysburd, and D. Karr. Building adaptive systems using Ensemble. *Software Practice and Experience*, 28(9):963–979, Jul 1998.

[51] R. v. Renesse, K. Birman, and S. Maffeis. Horus, a flexible group communication system. *Communications of the ACM*, 39(4):76–83, Apr 1996.

[52] E. Rescorla and A. Schiffman. The secure hypertext transfer protocol. Internet-draft, Terisa Systems, Inc., Jun 1998.

[53] L. Ricciulli, S. De Capatini di Vimercati, P. Lincoln, and P. Samarati. PNNI global routing infrastructure protection. In *Proceedings of the Information Survivability Workshop 1998*, pages 123–126, Orlando, FL, Oct 1998.

[54] O. Rodeh, K. Birman, M. Hayden, Z. Xiao, and D. Dolev. The architecture and performance of security protocols in the Ensemble group communication system. Technical Report TR98-1703, Department of Computer Science, Cornell University, Dec 1998.

[55] N. Sammons. Multi-platform interrogation and reporting with rscan. In *Proceedings of Ninth Systems Administration Conference (LISA '95)*, Monterey, CA, Sep 1995.

[56] D. Schmidt, D. Box, and T. Suda. ADAPTIVE: A dynamically assembled protocol transformation, integration, and evaluation environment. *Concurrency: Practice and Experience*, 5(4):269–286, Jun 1993.

[57] S. Staniford-Chen, B. Tung, and D. Schnackenberg. The common intrusion detection framework (CIDF). In *Proceedings of the Information Survivability Workshop 1998*, pages 139–143, Orlando, FL, Oct 1998.

[58] R. Thomas and R. Feiertag. Addressing survivability in the composable replaceable security services infrastructure. In *Proceedings of the Information Survivability Workshop 1998*, pages 159–162, Orlando, FL, Oct 1998.

[59] F. Travostino, E. Menze, and F. Reynolds. Paths: Programming with system resources in support of real-time distributed applications. In *Proceedings of the IEEE Workshop on Object-Oriented Real-Time Dependable Systems*, Feb 1996.

[60] W. Venema. TCP WRAPPER: Network monitoring, access control, and booby traps. In *Proceedings of the 3rd Usenix UNIX Security Symposium*, pages 85–92, Baltimore,MD, Sep 1992.

[61] J. Voas, G. McGraw, and A. Ghosh. Reducing uncertainty about survivability. In *Proceedings of the 1997 Information Survivability Workshop*, Feb 1997.

[62] A. Wespi, M. Dacier, H. Debar, and M. Nassehi. Audit trail pattern analysis for detecting suspicious process behavior. In *Proceedings of the 1st International Workshop on the Recent Advances on Intrusion Detection*, Louvain-la-Neuve, Belgium, Sep 1998.