Building Survivable Services Using Redundancy and Adaptation

Matti A. Hiltunen, Member, IEEE Computer Society, Richard D. Schlichting, Fellow, IEEE, and Carlos A. Ugarte, Student Member, IEEE

Abstract—Survivable systems—that is, systems that can continue to provide service despite failures, intrusions, and other threats—are increasingly needed in a wide variety of civilian and military application areas. As a step toward realizing such systems, this paper advocates the use of redundancy and adaptation to build survivable services that can provide core functionality for implementing survivability in networked environments. An approach to building such services using these techniques is described and a concrete example involving a survivable communication service is given. This service is based on Cactus, a system for building highly configurable network protocols that offers the flexibility needed to easily add redundant and adaptive components. Initial performance results for a prototype implementation of the communication service built using Cactus/C 2.1 running on Linux are also given.

Index Terms—Survivability, dependability, trustworthiness, redundancy, adaptation, intrusion tolerance, distributed systems.

1 Introduction

A survivable system is one that is able to continue providing service in a timely manner even if significant portions are incapacitated by attacks or accidents [4]. The challenges in building such systems are significant, especially if they are part of a large public network such as the Internet. In addition to having to deal with network and machine failures, a survivable system must have facilities to protect against threats and intrusions of different types, to detect intrusions when they occur, and to react to intrusions and repair damage. As such, survivability builds on research in security, reliability, fault tolerance, safety, and availability, as well as the combination and interaction of these different properties [53]. Note that, while we use the term “survivability” to emphasize certain system attributes, either “dependability” [40] or “trustworthiness” [51] could also be used.

This paper focuses on using the key enabling techniques of redundancy and adaptation to build survivable services that provide core functionality for implementing survivability in a networked environment. Such a service may provide, for instance, survivable or intrusion tolerant interprocess communication or data storage. Redundancy involves using extra resources to reduce the chance that an incident will compromise the entire system and can be used for data, communication, or in the form of application of multiple security techniques. For example, for a survivable communication service, redundancy might involve implementing message integrity by calculating redundant independent signatures or implementing confidentiality by encrypting the message with a combination of algorithms with keys established using different methods. Adaptation is the ability of software to modify its behavior at runtime and can be used as a technique to react to a suspected intrusion or to change execution unpredictably to complicate attacks. For example, for a communication service, adaptation might involve replacing a compromised encryption or key distribution method with a functionally equivalent method or periodically changing the keys used for existing secure channels. These techniques are then used in combination with other techniques such as intrusion detection [18], [44], data dispersion [7], [20], [23], [39], firewalls [12], and deception [14] to construct a survivable system.

The primary goal of this paper is to present an approach to building survivable services based on redundancy and adaptation. To do so, we first describe various ways in which survivability can be enhanced by using these techniques. We then illustrate the approach by giving a concrete example—the construction of a survivable communication service using a secure service called SecComm as a starting point. SecComm is a highly configurable communication service based on Cactus [33] that provides various alternatives for security properties such as privacy, authenticity, and integrity. This example also relates to our secondary goal of highlighting the type of system support needed to build a service of this type. In this case, the flexibility afforded by a system such as Cactus greatly simplifies the task of adding redundant and adaptive components to SecComm. Performance results from a prototype implementation built using Cactus/C 2.1 running on Linux are given to quantify the cost of survivability in this context.

M.A. Hiltunen and R.D. Schlichting are with AT&T Labs-Research, 180 Park Ave., Florham Park, NJ 07932.
E-mail: [hiltunen, rick]@research.att.com.
C.A. Ugarte is with the Department of Computer Science, University of Arizona, Tucson, AZ 85721. E-mail: cau@cs.arizona.edu.

Manuscript received 1 Feb. 2002; revised 7 Sept. 2002; accepted 20 Sept. 2002. For information on obtaining reprints of this article, please send e-mail to: te@computer.org, and reference IEEECS Log Number T17445.
2 USING REDUNDANCY AND ADAPTATION

2.1 Overview

A survivable service designed to operate across multiple hosts in a network must continue to implement some or all of its functionality despite external impacts, such as failures or intrusions, and while maintaining security guarantees, such as confidentiality and integrity. Survivability has many parallels with fault tolerance and, as such, many of the techniques used for fault tolerance can be adapted for survivability. With fault tolerance, the assumption is that any component can fail and the goal is to increase the probability that a system as a whole can continue to operate correctly despite the failure. Similarly, with survivability, the assumption is that any security mechanism can be compromised and the goal is to increase the probability that a system as a whole can continue to operate in an uncompromised fashion despite the attack.

This section explores the use of redundancy and adaptation—both techniques long associated with fault tolerance—in the context of survivability. Of course, as mentioned above, these techniques alone do not solve the problem, but rather should be used in combination with other techniques to develop a comprehensive solution.

2.2 Redundancy

Redundancy in fault tolerance usually takes the form of either space redundancy, such as replication of data or computation, or time redundancy, such as repeated execution or repeated message transmission. Redundancy that has proven useful for survivability includes techniques such as layered protection and data fragmentation and replication. An application of layered protection may involve, for example, encrypting critical files, which can protect data from an intruder even if the operating system itself is compromised [25]. The technique of data fragmentation and replication, which can improve availability as well as protect data from intruders, has been used in numerous systems [20, 23, 26, 39]. Threshold cryptography [19], in which a key is fragmented and the fragments replicated so that any k out of n fragments are required to access encrypted information, is based on a similar idea.

Another type of redundancy that has been explored less is the use of redundant methods. The basic idea is simple—with redundant methods enforcing a given security attribute such as privacy or integrity, the attribute should remain valid if at least one of the methods remains uncompromised. For example, to tolerate an attack against a public key-based PKI, a service might use two completely different authentication mechanisms (e.g., PKI and Kerberos). The same principle can be applied to communication security, where a message can be encrypted using combinations of different methods and signed using different signature algorithms. The value of using multiple different methods in this way is that it introduces diversity and can reduce the dangers associated with a vulnerability in any given single method that might be exploited by a systematic attack across multiple machines.

Redundancy techniques can be used to realize a number of different survivability principles. For example, redundancy helps avoid single points of vulnerability, including vulnerabilities and weaknesses in distributed security algorithms. The use of different combinations of redundant methods also introduces artificial diversity and unpredictability, which reduces the chances that an attack will be successful. In short, redundancy can help increase the survivability of a service by increasing the probability that it remains operational with appropriate security guarantees despite portions being compromised.

While redundancy can be a useful survivability technique, as with fault tolerance, its effectiveness depends on the details of how it is used. One important goal is maximizing the independence of the redundant elements, where two elements A and B are independent if compromising A provides no information that makes it easier to compromise B and vice versa. For example, if two independent methods, \( m_1 \) and \( m_2 \), are used to authenticate a user, breaking \( m_1 \) does not make it easier to break \( m_2 \). A simple example of nonindependence is when two encryption methods use the same key since, if one method is broken or the key stolen, privacy is compromised. This type of independence is very much analogous to the fault tolerance concept of independent failure modes for redundant hardware or software components. Components are independent in this sense when the failure of one component does not affect the correct execution of any other component.

Like fault tolerance, the idea of maximizing independence applies to many different aspects of a system's security architecture. In addition to affecting the choice of keys or passwords as described above, it applies to the use of cryptographic techniques, to the location in which different keys are stored, to the methods used for creating and distributing keys, and even to the choice of which parts of a system different administrators can access. Determining the degree of independence to use in each case depends on a cost-benefit analysis, that is, ascertaining how expensive it is to increase independence versus how likely it is that a particular aspect of the architecture will be attacked.

Another factor that determines effectiveness is the way in which the redundant elements are combined. For example, assume the goal is to protect a file using redundant encryption and two possible approaches are considered. In approach A, the first half of the file is encrypted using one method and the other half using the other method, while, in approach B, the whole file is encrypted using both methods sequentially. With approach A, an intruder can break the halves independently and knows when each method has been compromised, so the total effort required is proportional to the sum of the efforts required to break each method. On the other hand, with approach B, an intruder does not know whether either method has been compromised until both have been broken, so the effort required is proportional to the product of the efforts needed to break each method. Note, however, that using techniques in combination can often have unexpected subtleties. In the case of multiple encryption, for instance, if the methods form an algebraic group, breaking the double encryption is as easy as breaking single encryption [9]. In addition, more sophisticated attacks can be used to break multiple encryption faster than brute force attacks, such as
the “meet in the middle” attack used against double DES (2DES) [42].

While the cost of redundancy can typically be measured experimentally, the benefit—i.e., the increase in survivability—is much harder to quantify. Unlike fault tolerance, where the failure rate of a component can be measured and used to determine the necessary level of redundancy to achieve a given reliability, with survivability the attack rates vary unpredictably and adversaries constantly develop new attack techniques that can invalidate previous assumptions about the level of security provided by specific techniques. Indeed, quantifying the survivability of a system requires solving a number of difficult research challenges beyond the scope of this paper, many of which are the focus of research efforts elsewhere. These include quantifying attacker behavior [37], analysis of the reliability, latency, and cost-benefit for a system given that the probabilities for failures and successful intrusions are known for each component [36], model-based quantification of survivability metrics [50], and analysis of the security achievable using multiple cryptographic techniques [2].

The use of redundancy—especially method redundancy—to enhance survivability is explored further in the context of a survivable communication service in Section 3. In this service, the ability of Cactus to configure modules in flexible ways is exploited to allow redundant methods to be used to ensure such security attributes as privacy, authenticity, and integrity.

### 2.3 Adaptation

Adaptive software, i.e., software that changes its behavior at runtime, has been used in a number of contexts. For example, the Transmission Control Protocol (TCP) of the Internet protocol suite uses adaptive mechanisms for flow control, retransmission, and congestion control [35]. Other examples include adaptive media access control [13], adaptive encoding and compression [24], and adaptive routing algorithms [5]. Adaptation has typically been limited to changing parameter values such as timeout or transmission window size, but more recent work addresses algorithmic adaptations where a software component changes the algorithms used to implement its service at runtime [11], [47]. Many fault tolerance techniques can be viewed as specialized forms of adaptive behavior [10], [27], [31].

Like redundancy, adaptation is a useful technique for survivable systems and can be used to realize a number of survivability principles. For example:

- A service such as a communication protocol that changes its behavior deterministically increases artificial diversity, while a service that adapts non-deterministically can also increase its unpredictability.
- Adaptation can be used to implement graceful degradation; for example, if portions of the service have been compromised by an intrusion, the service may be able to adapt to an operating mode that excludes those portions but still satisfies some level of client requirements.
- Adaptation can be used to deal with changes in security and survivability requirements, as well as to respond to detected or suspected attacks; an example of the former is when stronger encryption may be required, while an example of the latter is terminating a suspected connection [8].

Coordination among software components on a given host or across hosts might be required, depending on the specific context. For example, if the encryption algorithm used in group communication is changed, all group members must be notified and the switchover synchronized so that no messages are lost. Such coordination requires special protocols [11], [47].

The adaptation process itself can be characterized by a general multiphase framework independent of the context in which it is used and the specific actions taken [31]. Specifically, the process is divided into three phases: change detection, agreement, and action. Change detection involves detecting a condition that might require an adaptation. Agreement follows change detection and involves reaching an agreement among the participants of a service on whether an adaptation should be made and what the adaptation should be. The action phase involves performing the adaptation itself—i.e., changing parameter values or algorithms—and includes any coordination that might be required. Note that agreement is not required if the adaptation only involves one host or if each host can adapt independently.

In addition to clarifying the process, these three phases also provide a framework for structuring an adaptive service and for implementing portions of the process as reusable software components. For example, the agreement phase and the coordination required in the action phase can often be implemented as algorithms that can be used in different contexts. The example communication service presented in Section 3 is structured in this way.

Using adaptation techniques as the basis for survivable services requires addressing a number of issues. Perhaps the most important problem is ensuring that the adaptation mechanism itself does not make the service more vulnerable. That is, the mechanism must prevent an intruder from compromising the service by having it deactivate security mechanisms or switch to weaker security mechanisms. For example, suppose that an adaptation mechanism is used for graceful degradation where the system switches to less costly encryption when resources are lost. In this scenario, an intruder may be able to trigger this adaptation by attacking those resources and then gain access to the system by breaking the weaker encryption. This particular problem can be resolved by not weakening security as part of a graceful degradation, but the issue is general. As was the case with redundancy, quantifying the increase in survivability afforded by adaptation is an open research issue that is beyond the scope of this paper.

Like redundancy, adaptation is a core technique used in the survivable communication service example in Section 3. In this case, the ability of Cactus to change execution patterns and module configurations dynamically allows the service to adapt in unpredictable ways and to react to external events.
3 COMMUNICATION: FROM SECURE TO SURVIVABLE

As an example, we demonstrate how the approaches outlined in the previous section can be applied to a secure communication service called SecComm. SecComm provides customizable communication security by allowing its user to choose which security attributes are required for a connection and which algorithms are used to implement these attributes. After giving an overview of SecComm, we describe how the service can be augmented with redundancy and adaptation techniques to make it more survivable.

3.1 SecComm Overview

SecComm is a highly configurable secure communication service with the inherent flexibility needed to realize redundancy and adaptation-based survivability techniques. We assume a typical distributed computing scenario consisting of a collection of machines connected by a local or wide-area communication network. Application-level processes communicate by using a communication subsystem that typically consists of IP, some transport-level protocol such as TCP or UDP, and, potentially, some middleware-level protocols. SecComm provides the abstraction of a bidirectional point-to-point communication channel for each connection that is opened.

SecComm uses established algorithms to provide security guarantees for the following properties:

- **Authenticity.** Ensures that a receiver can be certain of the identity of the message sender. Can be implemented using public key cryptography, any shared secret, or a trusted intermediary such as Kerberos.
- **Privacy.** Ensures that only the intended receiver of a message is able to interpret the contents. Can be implemented using any shared secret, public key cryptography, or combinations of methods.
- **Integrity.** Ensures that the receiver of a message can be certain that the message contents have not been modified during transit. Some authenticity and privacy methods also provide integrity as a side effect if the message format has enough redundancy to detect violations. Additional redundancy can be provided using message digest algorithms such as MD5 [48]. Integrity can be provided without privacy, but, at a minimum, the message digest itself must be protected.
- **Nonrepudiation.** Ensures that a receiver can be assured that the sender cannot later deny having sent the message. Relies on authenticity provided by public key cryptography and requires that the receiver store the encrypted message as proof.
- **Key distribution.** The keys needed by previous methods can be established in several ways. The user can distribute keys manually and have the application pass the keys to SecComm. Alternatively, SecComm can establish the keys itself, making use of protocols such as Diffie-Hellman [22] or external Certification Authorities and Key Distribution Centers such as Kerberos [43], [52].
- **Replay prevention.** Prevents an intruder from gaining an advantage by retransmitting old messages. Can be implemented using timestamps, sequence numbers, or other such nonces in messages. Typically used in conjunction with authenticity, privacy, or integrity since, otherwise, it would be trivial for an intruder to generate a new message that appears to be valid.
- **Known plain text attack prevention.** Prevents an intruder from utilizing known plain text-based attacks by including additional random information (“salt”) at the beginning of a message.

Each of these properties is associated with one or more software modules that implement the property using different algorithms. SecComm is configurable in the sense that specific modules can be selected based on user and application requirements. Such configurability is useful on two levels. First, it allows the user to determine which properties to use—if a certain property is not required or if it is already implemented by some other system layer, then no module that implements the property needs to be included. Second, it allows the user to select which module to use to implement a particular property. The user can evaluate the modules by various metrics, including the relative level of security and resource utilization.

All modules are syntactically independent from one another by virtue of the execution model supported by Cactus, which means that any combination is syntactically valid. However, the different security properties have certain semantic dependencies and ordering constraints that are reflected as equivalent semantic dependencies and ordering constraints between the modules that implement these properties. These dependencies can be recorded as a configuration graph that can be used to ensure any chosen configuration is semantically consistent [29]. Configuration constraints in SecComm are discussed in Section 4.2.

3.2 Redundancy

Security services that provide attributes such as privacy, integrity, and authenticity typically implement each attribute using a single method. For example, in a secure communication service, privacy may be provided by DES and integrity by keyed MD5. In choosing which algorithm to use to satisfy a given security property (e.g., privacy), one normally bases the selection on the tradeoff between an acceptable level of security and an acceptable cost. If more security is deemed necessary, a better algorithm or better version of the same algorithm (e.g., by using a longer key) is chosen; this often results in greater resource utilization.

Although such an approach may be secure in the traditional sense, it is not survivable—once a method is compromised, all security guarantees on the connection related to that attribute are gone. Note that a security method has multiple different points of attack. In the case of secure communication, the encryption algorithm may be broken, the key may be stolen from a user or user’s machine, or the key distribution method may be broken and, thus, the key assigned to a connection may be known by an intruder. Each method has, in essence, multiple single points of vulnerability very much analogous to a single
point of system failure when considering fault tolerance attributes. This problem is the same for many other aspects of security, including authentication and access control.

The redundancy techniques described in Section 2 can be used to enhance the survivability of SecComm. These techniques can be used within a single SecComm connection or by creating multiple redundant connections. Within a connection, this is done by using two or more techniques to guarantee an attribute rather than a single method. Given that, an attack may be successful against one method, but the system itself will remain secure if the other methods remain uncompromised. For maximal independence, each method should use a separate key established using different key distribution methods to avoid an attack based on stealing keys or compromising a key distribution method. If multiple SecComm connections are created, the data sent can be fragmented between the connections and different combinations of methods can be used for the different connections. In this case, the SecComm connections can also be established between different pairs of machines and the message data fragmented so that even the full compromise of one machine does not reveal the information or make it impossible to transmit the information. Finally, these machines should be diverse enough that it is unlikely they can be compromised using the same attack.

Here, we focus on redundancy within a single connection since extending the ideas to multiple connections is straightforward. The simplest way to make use of this redundancy is to apply the different methods successively on the same data. For example, a message might first be encrypted using DES and then AES [17]. However, there are many other approaches, including:

- Alternating the order in which methods are applied, e.g., apply DES before AES for some messages and AES before DES for others.
- Applying different methods to different parts of the data, e.g., encrypt different parts of one message or different messages in a stream using different methods.

The first approach is static since every message is processed by the same methods, while the other two are dynamic since different methods can be used in different ways for different messages. The resulting unpredictability is intended to make an adversary’s task more difficult.

Maximizing the independence of the methods and keys used within a single SecComm connection or between multiple connections increases the survivability of the service. Given that different types of attacks can be mounted against a secure communication service, there are different aspects of independence. These include:

- Independence of keys, meaning that keys are different, are generated using different methods, are stored in different locations, and are administered by different administrators.
- Independence of methods, meaning that the methods used are diverse, both within a connection and across connections.

For example, if an attack attempts to determine a key by guessing, then using two independent keys to secure a connection is sufficient to increase survivability. On the other hand, if the attack exploits a weakness in the algorithm, it would be necessary to use two different algorithms. Similarly, if the attack compromises one of the machines that serves as the endpoint of the secure connection, it would be necessary to use connections between multiple pairs of diverse machines to minimize the chance of the attack compromising both.

In the case of encryption methods, independence is difficult to argue rigorously, but the risk of methods not being independent is likely to be minimized if the methods are substantially different or if they encrypt data in different size blocks. It is also possible to develop combinations that attempt to maximize independence by not simply encrypting the same data multiple times, but by combining the methods in different ways. For example, suppose that $m$ is a cleartext message and $E_1$ and $E_2$ are different encryption methods. A ciphertext message $cm$ could be constructed as $cm = \{E_1(m \oplus r), E_2(r)\}$, where $\oplus$ is the exclusive-or operation and $r$ is a random bit sequence the same length as the message. The message sender generates $r$ individually for each message sent. The receiver can reconstruct $m$ by decrypting the parts of the message using $E_1$ and $E_2$, respectively, and then performing exclusive-or operation between the parts to eliminate $r$. Given this method, breaking only $E_1$ or $E_2$ does not produce any useful information, which means that the attacker has to break both simultaneously to compromise the system. As a result, the effort required is multiplicative. Note that the random bit sequence should be truly random or at least difficult to guess or, otherwise, the attacker may not need to break $E_2$.

Determining independence of methods is easier for other security attributes such as message integrity. Let $m$ be the message to be protected and $d_1(m), d_2(m), \ldots$ be different cryptographic message digests of $m$. Since the message digest algorithms operate on the message independently, an attacker would need to compromise each integrity algorithm separately. In this case, the increase in the breaking effort is additive since the attacker knows when each method has been broken.

### 3.3 Adaptation

Fine-grained configurability allows SecComm to be tailored to the needs of an application and the execution environment. In many cases, these needs are not static, but rather can change at runtime. For example, the quality and security of a network connection may change as a mobile user changes location. It may also be necessary to change security algorithms that are suspected to have been compromised and introduce new security algorithms to strengthen the system. Adaptation can be applied in these types of situations to increase system survivability.

Adaptation in SecComm can be seen as dynamic configurability of methods augmented with adaptation algorithms to handle the adaptation process. The same library of methods that makes the system configurable is available so that the best method—or combination of methods—can be chosen based on the current needs. The choice of configuration can be made by the application or by the adaptation algorithm. The adaptation algorithm must also handle any monitoring for changes that might
lead to an adaptation, any negotiation that might be necessary, and the actual switchover from the old configuration to the new configuration. The current version of SecComm performs adaptation between methods that have been linked statically into the service. While it is possible to load new methods at runtime using dynamic libraries, such an approach has additional security risks that are not addressed in this paper.

As noted above, the first phase in an adaptation is change detection, which triggers the process. The change may occur for any number of different reasons, including:

- **Availability of local resources.** As processor and memory utilization increases, it may be beneficial to change to an algorithm that uses less resources. Similarly, if processor and memory utilization decreases, it may be beneficial to change to a more effective algorithm that uses more resources.
- **Availability of network resources.** Bandwidth and latency may change, prompting the switch to a different algorithm. Routing changes may also trigger adaptations.
- **Intrusion detection.** When an intrusion is detected by, for example, an intrusion detection system (IDS), the service may adapt by replacing existing security algorithms with stronger ones. Similarly, if the intruder has managed to compromise completely the service on a given host, any existing connections to that host may need to be terminated.
- **User directive.** The user may trigger an adaptation directly. The user’s options range from signaling that an adaptation is required to specifying what the new configuration should be.

The task of monitoring to detect changes can be done within SecComm as a separate module. Alternately, monitoring can be done externally and a signal sent to SecComm when an appropriate event is detected.

The second phase is agreement. Within SecComm, there is an evaluation module that determines when an adaptation is needed and what the new configuration should be. In simple adaptations, this decision can be made by a single host and no agreement is needed. In more complex cases, agreement between both hosts may be required. In SecComm, the host that initiates the adaptation also coordinates the agreement. This host is currently assigned statically for each connection, although it could, in principle, be done dynamically.

The third phase is to perform the adaptive action, i.e., to switch the algorithms. One simple approach is as follows: The initiating host signals the other host that an adaptation is about to occur, stops transmitting messages with the old configuration and awaits confirmation from the second host. Upon reception of the initiator’s message, the second host deactivates the old configuration, activates the new configuration, and transmits an acknowledgment message. Once this acknowledgment is received, the initiating host switches from receiving messages using the old configuration to receiving messages using the new configuration.

It is important that no messages are lost during the transition, so the initiating host stops transmitting messages until it is certain that the second host is ready to accept messages using the new configuration. Before it sends the acknowledgment, the second host uses the old configuration to send messages; the initiating host keeps the old configuration active only for message reception. Note that this scheme relies on the underlying protocols to provide reliable ordered delivery. If either of these properties is not guaranteed, a different protocol must be used.

One of the main concerns in adding a new module to a secure or survivable system is that no new vulnerabilities be introduced. With respect to adaptations, some of the possible vulnerabilities are:

- **Providing information to an adversary.** An intruder should not be able to examine the adaptation messages and benefit from it.
- **Triggering adaptations.** An intruder should not be able to trigger an adaptation.
- **Denial of service.** An intruder should not be able to prevent an adaptation from completing.

Techniques for addressing these issues in SecComm are discussed in Section 4.

4 IMPLEMENTATION

The traditional security methods offered by SecComm can be combined with techniques that offer redundancy and adaptation to produce a version of the service that is more survivable. Here, we discuss implementation aspects, focusing on how the abstract methods discussed above map into concrete components. The basis for this is Cactus, which allows these components to remain independent, yet interact with each other as needed. We first give an overview of Cactus, then explain in detail how SecComm makes use of these features.

4.1 Cactus

Cactus is a system for constructing configurable network protocols and services where each service property or functional component is implemented as a separate software module called a microprotocol [33]. A customized instance of a service is then created by choosing a collection of microprotocols based on the properties to be enforced and configuring them together with the Cactus runtime system to form a composite protocol that implements the service on each machine. A microprotocol is structured as a collection of event handlers that are executed when a specified event occurs. Events can be raised explicitly by microprotocols or by the Cactus runtime.

The primary event-handling operations are:

- **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. Handler execution is atomic with respect to concurrency, i.e., a handler is executed to completion before execution of any other handler is started unless the handler voluntarily yields execution by either raising another event synchronously or by invoking a blocking semaphore operation. In the case of a synchronous raise, the handlers bound to the raised event are executed before control returns to the handler that issued the raise. In addition to the flexible event mechanism, Cactus supports shared data that can be accessed by all microprotocols configured into a composite protocol.

Finally, Cactus provides a message abstraction, called `dynamic messages`, that is designed to facilitate development of configurable services. The main features provided by dynamic messages are named message attributes and a coordination mechanism that only allows a message to be transferred out of a composite protocol when agreed by all microprotocols. Message attributes are a generalization of traditional message headers and have scopes corresponding to a single composite protocol (`local`), all the protocols on a single machine (`stack`), and the peer protocols at the sender and receiver (`peer`). A customizable pack routine concatenates peer attributes to the message body for network transmission or for operations such as encryption and compression. A corresponding unpack routine extracts the peer attributes from a message at the receiver.

The flexibility of Cactus allows abstract service properties and functions to be implemented as independent modules without enforcing artificial ordering between modules as in hierarchical composition frameworks such as the x-kernel [34]. Furthermore, the indirection provided by the event mechanism makes it easy to change the collection of microprotocols dynamically without affecting other microprotocols.

The facilities provided by Cactus are not tied to any specific programming language, architecture, or operating system. Several prototype implementations of Cactus have been constructed, including versions written in C, C++, and Java, running on Linux, Solaris, and other platforms. In addition to SecComm, other prototype services that have been successfully implemented using Cactus or the predecessor Coyote system [6] include group RPC [30], membership [32], and a real-time channel abstraction [33].

### 4.2 SecComm Implementation

SecComm executes in user space with either IP, UDP, or TCP as the underlying protocol. As described above, SecComm allows fine-grain customization of a range of security attributes including privacy, authenticity, message integrity, replay prevention, nonrepudiation, and key distribution.

![Fig. 1. Microprotocol classes and interactions.](image)

#### 4.2.1 Microprotocol Structure

The abstract security attributes and key distribution are implemented by one or more microprotocols. When a number of microprotocols implement variations of the same abstract property, we collectively refer to them as a class of microprotocols. For example, the class of privacy microprotocols includes DESPrivacy, RSAPrivacy, and IDEAPrivacy microprotocols that use the DES, RSA, and IDEA algorithms, respectively. Fig. 1 illustrates the main microprotocol classes and typical event interactions between them.

The design of the SecComm service allows any combination of security microprotocols to be used together in both static and dynamic ways. The ability to use multiple microprotocols within a given class at the same time is one way in which redundancy can be used to support a survivable service. Naturally, there may be configuration constraints between microprotocols that affect which combinations are feasible.

SecComm has four major types of microprotocols: basic security microprotocols that perform simple security transformations such as encryption or integrity checks, key distribution microprotocols that allow the safe exchange of keys used by other microprotocols, meta-security microprotocols that build more complex security protocols using the basic security microprotocols as building blocks, and adaptation microprotocols that allow the configuration of existing basic and meta-security microprotocols to be changed dynamically. An example of a simple security microprotocol is DESPrivacy, which provides privacy of data exchange using the DES algorithm. An example of a meta-security microprotocol is MultiSecurity, which uses multiple basic security microprotocols to provide stronger guarantees. An example of an adaptation microprotocol is SimpleAdaptation, which dynamically switches the microprotocol used to implement a given security property. Basic security and key distribution microprotocols are discussed further below, while meta-security and adaptation microprotocols are described in Sections 4.3 and 4.4, respectively.

Our prototype implementation of SecComm uses the Cryptlib cryptographic package [28] to provide basic cryptographic functionality. Any cryptobinary with the necessary functions could be used, however.
4.2.2 Application Programming Interface

SecComm allows a higher level service or application to open secure connections and then send and receive messages through these connections. The specific operations exported by SecComm are the following:

- **Open**(*participants, role, config*). Opens a session for a new communication connection, where *participants* is an array identifying the communicating principals, *role* identifies the role of the participant in opening the connection (active or passive), and *config* is a configuration that captures the desired security properties of the session.

- **Push**(*msg*). Passes a message from a higher-level protocol or application to a SecComm session to be transmitted with the appropriate security attributes to the participants.

- **Pop**(*msg*). Passes a message from a lower level protocol to a SecComm session to be decrypted, checked, and potentially delivered to a higher level protocol. When the SecComm protocol passes a message to the higher level and authentication is required, it adds a stack attribute that is the ID of the authenticated sender.

- **Close**(). Closes a SecComm communication session.

- **Adapt**(*config*). Triggers an adaptation. The new configuration is provided via the optional argument *config* or can be selected internally by the adaptation protocol.

We assume that the participants of the communication connection negotiate the properties for the connection on a higher level. Once negotiated, they are specified in the **Open** operation as two ordered lists of microprotocols and their arguments, the first for messages going downward through the composite protocol and the second for messages going upward. Thus, for example, the following specifies that messages going downward are processed first by DES and then by MD5, while messages going upward are processed by the same microprotocols but in the reverse order:

```plaintext
{DES(DESkey), MD5(MD5key); MD5(MD5key), DES(DESkey)}.  
```

Our eventual goal is to develop an approach in which properties are given as formal specifications that are then translated automatically into collections of microprotocols and arguments.

4.2.3 Basic Security Microprotocols

The basic security microprotocols are simple, typically consisting of two event handlers and an initialization section. One of the event handlers is used for the data passing down through the SecComm protocol and the other one is used for data passing up through the protocol. The initialization section of the microprotocol is executed when a new SecComm connection is opened, i.e., when a session is created.

A basic security microprotocol (Fig. 2) typically takes four or five arguments. In this parameter list, *dEvent* and *uEvent* are events that signify message arrival from an upper and lower-level protocol, respectively. The two handlers in the microprotocol are bound to these events to initiate execution at the appropriate time. The *dOrd* and *uOrd* parameters are the relative orders in which this particular security microprotocol is to be applied to messages flowing down and up, respectively. Some algorithms call for the use of keys; those basic security microprotocols that implement such algorithms allow for its optional specification via the *key* argument.

Note that if the key used by the security microprotocol has yet not been established, it raises an event **keyMiss** that is handled by the key distribution microprotocols. This event is raised synchronously and, thus, the handler is blocked until the associated event handlers have completed execution. This allows the key distribution microprotocols to block the appropriate handler until the key has been established.

The design uses event pointers as arguments rather than fixed event names to allow multiple types of configurations, an approach that demonstrates the inherent flexibility provided by an event-based execution model.

4.2.4 Key Distribution Microprotocols

If the keys used by the secret key cryptographic methods are not agreed upon a priori, they must be established after the communication session is opened. As with the other security properties, we use established algorithms to do key distribution. Each algorithm is implemented as a separate microprotocol and each basic security microprotocol can be associated with a different key distribution microprotocol. This association can be specified through the **Open** operation by the application; if omitted, the default key distribution microprotocol is selected.

All key distribution microprotocols support the following operations that are accessed indirectly through the event mechanism:

- **KeyRegister**(*key*). A basic security microprotocol that has been supplied with a key must register it with the key distribution microprotocol.

- **KeyMiss**(*key*). A basic security microprotocol that is missing a key invokes this operation. Upon completion, the *key* argument contains the necessary key.

By requiring that these two operations be provided, all basic security microprotocols can use the same interface. The

```plaintext
micro-protocol BasicSecurity(dEvent,dOrd,uEvent,uOrd,key) {
  handler ProcessDownMsg(msg){
    if myKey == NULL raise(keyMiss,myKey,SYNC);
    add attributes, pack, encrypt, etc.;
  }
  handler ProcessUpMsg(msg){
    if myKey == NULL raise(keyMiss,myKey,SYNC);
    decrypt, unpack, check attributes, etc.;
  }
  initial { myKey = key; 
    bind(dEvent,ProcessDownMsg,dOrd);
    bind(uEvent,ProcessUpMsg,uOrd);
  }
```

Fig. 2. Generic basic security microprotocol.

---

The content includes a code snippet demonstrating the usage of a basic security microprotocol named `BasicSecurity`. This snippet outlines the handler methods `ProcessDownMsg` and `ProcessUpMsg` which are triggered by events `dEvent` and `uEvent` respectively. Each handler method checks if the `myKey` is null, raises a `keyMiss` event if true, and performs operations such as adding attributes, packing, encrypting, and decrypting, unpacking, checking attributes, etc. The `initial` block binds the event handlers to the appropriate messages, indicating the order of operation for message reception.
other operations—the ones that implement the bulk of the work—can differ from one key distribution microprotocol to the next. We classify key distribution microprotocols based on who is responsible for generating the key; among the potential options are:

- **Asymmetric.** One communicating principal (e.g., a client or a server) creates a session key and distributes it to the other principals.
- **Symmetric.** A session key is created using the Diffie-Hellman algorithm.
- **External.** Some external security principal creates the session key and distributes it to communicating principals (e.g., Kerberos, certification authority).

Note that whether the key distribution protocol is symmetric or asymmetric is orthogonal to the type of encryption method for which they keys are used and, in particular, whether the encryption method itself is symmetric (e.g., DES) or asymmetric (e.g., RSA).

### 4.2.5 Configuration Constraints

A number of factors must be considered when microprotocols are combined into a custom instance of the SecComm service. In particular, there are both algorithmic or property-based constraints that are independent of a particular implementation and implementation constraints that are specific to our Cactus-based prototype. Algorithmic constraints are those that result from the inherent nature of the properties being enforced or the algorithms used. For example, the nonrepudiation microprotocol requires the use of an authenticity microprotocol based on public keys. Similarly, all microprotocols that use a key require either that the key is provided when the session is created or that a key distribution microprotocol is included.

Other algorithmic constraints affect the order in which various security algorithms are applied. For example, all attack prevention microprotocols should execute before privacy, integrity, or authenticity microprotocols at the sender to ensure that the mechanism used for attack prevention is protected from modification. Similarly, nonrepudiation microprotocols should be executed immediately before authentication at the receiver so that only the sender’s public key is required to later prove the message was sent by the sender. Other ordering constraints have been identified elsewhere [1], [3].

Implementation constraints are those that result from the specific design of the SecComm microprotocols. Compared with systems that support linear or hierarchical composition models, the nonhierarchical model supported by Cactus introduces minimal implementation constraints on configurability. That is, with Cactus, it is generally possible to implement independent service properties so that this independence is maintained in the microprotocol realization. When extra constraints do get imposed, it is usually because making an extra assumption about which other microprotocols are present significantly simplifies the implementation.

In the current SecComm prototype, the only additional implementation constraint is that each integrity and replay prevention microprotocol can be used at most once in a given configuration. Thus, for example, two instances of

```plaintext
micro-protocol MetaSecurity(dEvnt,dOrd,uEvnt,uOrd, downBasicEvnts,upBasicEvnts) {
    handler ProcessDownMsg(msg){
        in some order raise(downBasicEvnts[i],msg,SYNC);
    }
    handler ProcessUpMsg(msg){
        in some order raise(upBasicEvnts[i],msg,SYNC);
    }
    initial {
        bind(dEvnt,ProcessDownMsg,dOrd);
        bind(uEvnt,ProcessUpMsg,uOrd);
    }
}
```

Fig. 3. Generic meta-security microprotocol.

MD5Integrity cannot be used together, while MD5Integrity and SHAIntegrity can be. This restriction results from the use of fixed message attribute names for each microprotocol, which could be avoided by dynamically assigning attribute names at startup time.

### 4.3 Redundancy Using Meta-Security Microprotocols

The basic SecComm design supports redundancy implicitly since multiple microprotocols for a given security property can be included in the same service. For example, multiple encryption microprotocols can be configured together to provide privacy through redundant methods. However, the meta-security microprotocols have been designed specifically for implementing more complex redundancy schemes by allowing basic security microprotocols to be combined in sophisticated ways. For example, a meta-security microprotocol may apply multiple or alternating basic security microprotocols to a message.

The basic structure of a meta-security microprotocol is shown in Fig. 3. In this design, the microprotocol is passed vectors of down and up events that correspond to the events to which handlers in the basic microprotocols have been bound as arguments `downBasicEvnts` and `upBasicEvnts`.

Examples of different specific meta-security microprotocols include:

- **MultiSecurity.** Applies multiple basic security protocols to a message in sequence.
- **AllSecurity.** Applies one security microprotocol to each message, with the method chosen successively from a specified list. If the sequence of methods is deterministic or agreed upon by the sender and receiver, no additional information is required provided that the underlying communication is reliable and maintains FIFO ordering.
- **RandomAltSecurity.** Similar to AllSecurity but uses a randomly chosen method for each message. Each message must carry an identifier than can be used by the receiver to determine which method to use to decrypt the message.
- **ExpansionSecurity.** Uses the technique mentioned in Section 3.2 that xors the message body with a
random bit sequence and encrypts the result (part one) and the random bit sequence (part two) with given basic security microprotocols.

A meta-security microprotocol can also be configured to use other meta-security microprotocols. For example, we can implement AltMultiSecurity that applies alternating different multiple encryption methods to each message by combining AltSecurity with MultiSecurity.

The concept of meta-security microprotocols can be applied to increase the survivability of any security property for which using multiple or alternating methods reduces the chance of successful attack. Privacy, authenticity, and message integrity, among others, fall in this category. The SecComm design does not prevent the same idea from being used for other properties such as replay prevention and nonrepudiation, but the benefit for such properties is more questionable. Finally, note that the ease with which such meta-security microprotocols can be constructed is again a direct result of flexibility provided by Cactus.

Key distribution has security risks analogous to data communication, but with greater potential impact since the compromised key will likely be used for a period of time. Thus, the same redundancy techniques used for data security can also be applied for key distribution security. Multiple key distribution microprotocols can also be used to obtain keys redundantly.

Redundancy and key distribution can mix in different ways. For example, relying on redundant trusted arbitrators to obtain a key in an external scheme can avoid some of the problems that occur if a single arbitrator is used and compromised. Moreover, if the multiple arbitrators are thought to be vulnerable to the same attack, different algorithms can be used.

In the above example, the multiple methods are used collaboratively to obtain the same key. The scheme can also be used to collect different keys, however. The simplest scenario has each key assigned to a separate basic security microprotocol. A more complex configuration would allocate multiple keys to the same microprotocol, which could use alternate keys on a message by message basis.

4.4 Adaptation Using Adaptation Microprotocols

The action phase of the adaptation process is implemented in SecComm by adaptation microprotocols in concert with adaptation-aware microprotocols, which are basic security microprotocols augmented with the ability to be activated and deactivated. An adaptation microprotocol handles both the local and remote aspects of adaptation. Local processing involves activating and deactivating the microprotocols associated with the old and new configurations on the local host. Activation is done by binding an event handler to the events associated with messages; deactivation removes such bindings. Activation and deactivation are specific to messages flowing in a given direction—either from the application to the network or vice versa—which allows finer-grain control over the process. The remote aspects of adaptation deal primarily with coordination. An adaptation microprotocol communicates with its peer on the remote host and initiates the exchange of messages that determines when individual microprotocols are activated or deactivated. The particular order depends on the goals of the adaptation.

SimpleAdaptation is an example adaptation microprotocol in which the order of activation and deactivation guarantees that all messages sent using the old configuration are delivered before any of the messages sent using the new configuration. The structure of this microprotocol follows the description in Section 3 and is shown in Fig. 4. Here, the originating host (master) first disables the old configuration for outgoing messages and informs the other host (slave) that an adaptation is occurring. Upon receipt of an adaptation message from the master, the slave sends an acknowledgment, disables the old configuration, and enables the new configuration. Upon receipt of the acknowledgment message from the slave, the master disables the old configuration for incoming messages and enables the new configuration for both incoming and outgoing messages. As noted above, this algorithm requires reliable ordered message delivery. A more symmetric group-oriented version of the protocol that requires more extensive coordination is presented in [11].

Other adaptation algorithms can be constructed to handle different requirements or environments. For example, message delivery might not be reliable or ordered, so the adaptation algorithm might have to retransmit control messages. Another adaptation algorithm might require some form of agreement between both hosts before any adaptation can take place.

Three vulnerabilities arising from adaptation methods were identified in Section 3. The first is the problem of potentially providing additional information to an adversary. For example, since the messages exchanged by the adaptation microprotocol can be distinguished from other messages, it is possible to determine that an adaptation is occurring. The contents of the message need not reveal
5 EXPERIMENTAL RESULTS

A prototype of SecComm with extensions for redundancy and adaptation has been implemented using the C version of Cactus on a cluster of 600 Mhz Pentium III PCs running Linux 2.4.7 connected by a 1 Gb Ethernet. This section provides performance results that illustrate the cost of redundancy and adaptation in SecComm.

The current prototype implements a subset of the microprotocols presented in this paper, including privacy microprotocols based on DES, RSA, IDEA, Blowfish, and XOR, integrity microprotocols based on MD5 and SHA, an authentication microprotocol based on DSA, a timestamp based replay prevention microprotocol, a nonrepudiation microprotocol, two meta-security microprotocols, and one adaptation protocol. Other microprotocols are currently being implemented.

We have conducted a number of experiments using different subsets of microprotocols. Table 1 gives roundtrip times (RTT) in microseconds and the throughput of SecComm for processing outgoing messages using different configurations. The RTT test used 100-byte messages, the throughput test used 1,400-byte messages, and the figures were computed over 1,000 or more roundtrips. The system was lightly loaded during testing and all SecComm configurations use IP as the underlying protocol, with no packet reordering or drops observed during the experiments. In these tests, DESPrivacy uses a 56-bit key running in CFB mode, BlowfishPrivacy uses a 448-bit key running in CFB mode, XORPrivacy uses a 64-bit “key,” and IDEAPrivacy uses a 128-bit key running in CFB mode. The NonRepudiation microprotocol tested ensures that messages are written to disk before the message is delivered to the next level. Other nonrepudiation variants that allow delayed write to disk are naturally less expensive. As a baseline, an average roundtrip time using IP directly on this cluster is 365 µs. The notation “+ microprotocol name” in the table indicates that the named microprotocol is added to the configuration on the previous row of the table.

The entry for base SecComm reflects times for a skeleton version of SecComm that does not use any microprotocols; its additional cost indicates the approximate cost of adding a new protocol to the stack. The cost over IP column indicates the roundtrip time overhead of the configuration compared to using just IP. Similarly, the cost over base column indicates the overhead of the configuration compared to just the base.

The cost over base column provides the most realistic indication of the cost of using redundancy techniques implemented as meta-security microprotocols. For MultiSecurity, these numbers indicate that the cost is roughly equal to the sum of the costs associated with the corresponding microprotocols. For example, the overhead of using MultiSecurity to combine DES and Blowfish is 483 µs, which is actually slightly less than the sum of the costs of DES and Blowfish since the cost of using Cactus mechanisms is amortized over multiple microprotocols. For AltSecurity, the cost is approximately the same as the average cost of the individual microprotocols.

The throughput test measured how much data an application could push through SecComm on the sending host. This experiment measured SecComm in isolation without including either the lower-level protocols (IP and Ethernet) or the network transmission. The measurements were performed by inserting an additional protocol between SecComm and IP that simply drops messages when a throughput test is performed. The base SecComm entry shows the highest achievable throughput, obtained when there is the least overhead and the message data is not inspected. The use of XOR illustrates the cost of using a single microprotocol that does minimal computation on
every byte of the message. The established algorithms perform significantly more work and reduce the throughput by greater amounts. The high throughput of NonRepudiation is due to the fact that most of the work of NonRepudiation is done on the receiving host and this test does not capture the throughput at the receiving host. Combining microprotocols for MultiSecurity results in lower throughput that is consistent with the throughput values of the individual microprotocols. For AltSecurity, the throughput is slightly less than the average throughput of the individual microprotocols.

We also measured adaptation time and the cost associated with supporting adaptive changes. The experiment involved a simple adaptation that switches between DES and XOR when the user gives an adaptation signal. The delay—which reflects the amount of time that elapsed on the initiating host from the time the adaptation is triggered until the time it completed—was measured at 442 µs. In these initial tests, adaptation messages were not encrypted or otherwise protected, but we anticipate that the additional overhead would be comparable to those given above. To estimate the overhead associated with supporting adaptation, we performed the RTT and throughput tests using a configuration where adaptation was possible but never triggered. In this configuration, the adaptation-aware DES microprotocol was active throughout the lifetime of the connection. The RTT was measured at 668 µs and the throughput at 50.28 Mb/s.

The experiments demonstrate two aspects of adaptation. First, the numbers indicate that the adaptation microprotocol has almost no overhead if the adaptation does not occur, i.e., the only cost is associated with the actual adaptation process. Second, the experiments in which adaptation does occur suggest that the process does not impose significant additional execution overhead or communication delay. Experiments that include protection of the adaptive process itself are continuing and are expected to allow precise quantification of this delay under more realistic conditions.

6 Related Work

The basic idea of using redundancy to increase the survivability of services in networked systems has been used elsewhere. For example, redundancy in the form of fragmentation and scattering has been used for intrusion-tolerant data storage [20], [25], [26]. Replication has also been used for authentication and authorization services [7]. Finally, redundancy in communication has been used in [15] in terms of sending each message along multiple disjoint paths from the sender to the receivers. Although this work does not explicitly address intrusions, the algorithm is designed to tolerate arbitrary failures and ensure message integrity and service availability.

In contrast with the above examples that use data or space redundancy, this paper has focused on using method redundancy. The combination of these two types of redundancies can provide an even higher level of survivability for distributed services. To our knowledge, the only similar services that use method redundancy are intrusion detection systems (IDS) [18]. If an IDS employs redundant detection modules with different detection algorithms, it stands a greater chance of detecting more intrusions and giving fewer false alarms.

The basic idea of adaptation has also been used in a number of services. Many intrusion-tolerant services adapt when intrusions are detected by terminating suspected connections or quarantining infected machines [8]. The ITU project [16] proposes using unpredictable adaptations to enhance survivability for replicated servers that use group communication. The unpredictable adaptations include starting new replicas in unpredictable locations and changing the replication policy of a replica group. Finally, although it does not specifically address survivability, [10] presents an adaptive version of [15], where the algorithm is switched to tolerate arbitrary failures only when such a failure is detected. To our knowledge, no other system performs algorithmic adaptations as a reaction to suspected intrusions.

Work specifically related to SecComm can be divided into secure communication standards and other configurable secure communication services. Some degree of customization is supported in several recent standards. For example, IPsec allows a choice of security options, including message integrity and privacy, using a selected cryptographic method [38]. It is also possible to apply multiple security methods to a given communication connection. TLS (Transport Level Security) [21] offers a choice of privacy (e.g., DES or RC4), integrity (e.g., keyed SHA or MD5), and optional message compression, but does not directly support the use of redundant methods. None of these protocols support runtime adaptation or provide flexible facilities to implement redundant methods as done in SecComm.

Configurable secure communication services have been implemented using various configuration frameworks, including the x-kernel [46], Ensemble [49], and the framework described in [45]. All these models are similar in the sense that a communication subsystem is constructed as a directed graph of protocol objects. Although this allows arbitrary combinations of security components, the structure is limiting compared to Cactus and would make it difficult to implement some of our more dynamic redundancy techniques. However, Antigone [41] has adopted an approach similar to Cactus in which microprotocols and composite protocols are used to implement secure group communication with customizable policies, including rekeying and message security. To our knowledge, none of these projects focus on using redundancy and adaptation to enhance survivability.

7 Conclusions

The use of redundancy and adaptation can increase the survivability of services for networked systems by making them tolerant to intrusions and other attacks and by allowing them to change execution behavior to increase unpredictability or to react to attacks. This paper has discussed the use of these two techniques and presented a concrete example that involves augmenting the SecComm secure communication service with redundant security methods and support for adaptation. This service has been constructed using Cactus, a system that provides the type of flexible interaction and
configuration mechanisms needed to build services of this type. Experimental results from a prototype implementation suggest that the performance of the service is proportional to the cost of its constituent methods.

Future work will include further experimentation with the survivable SecComm prototype and the development of a family of adaptation protocols with different execution characteristics. A special focus will be on designing adaptation protocols that are scalable and that minimize the synchronization required to support coordinated changes in execution behavior. We intend to experiment with these protocols for such uses as building location-specific mobile services, as well as within the context of survivability.

ACKNOWLEDGMENTS

Gary Wong implemented the Cactus framework used for the SecComm implementation. He also provided excellent comments and suggestions that improved the paper. Trevor Jim provided useful comments on an earlier version of the paper. This work was supported in part by the US Defense Advanced Research Projects Agency under grant N66001-97-C-8518 and the US National Science Foundation under grant ANI-9979438.

REFERENCES

Matt A. Hiltunen received the MS degree in computer science from the University of Helsinki in 1989 and the PhD degree in computer science from the University of Arizona in 1996. He is currently a senior technical staff member in the Dependable Distributed Computing Department at AT&T Labs-Research in Florham Park, New Jersey. He is a member of the ACM and the IEEE Computer Society. His current research interests include dependability, performance, timeliness, and security in distributed systems and networks.

Richard D. Schlichting received the BA degree in mathematics and history from the College of William and Mary and the MS and PhD degrees in computer science from Cornell University. He is currently head of the Dependable Distributed Computing Department at AT&T Labs-Research in Florham Park, New Jersey. He was on the faculty at the University of Arizona from 1982-2000 and spent sabbaticals in Japan in 1990 at the Tokyo Institute of Technology and in 1996-1997 at the Hitachi Central Research Lab. He is an ACM fellow, an IEEE fellow, and a member of IFIP Working Group 10.4 on Dependable Computing and Fault Tolerance. He is on the editorial board of the IEEE Transactions on Parallel and Distributed Systems and has been active in the IEEE Computer Society Technical Committee on Fault-Tolerant Computing, serving as chair from 1998-1999. His research interests include distributed systems, highly dependable computing, and networks.

Carlos A. Ugarte received the BS degree in computer science from the Georgia Institute of Technology and the MS degree in computer science from the University of Arizona. He is currently working toward the PhD degree in computer science at the University of Arizona. His research interests are in distributed systems and networks. He is a student member of the IEEE.

> For more information on this or any computing topic, please visit our Digital Library at http://computer.org/publications/dlib.