PROTECTING APPLICATIONS AGAINST MALICE USING ADAPTIVE MIDDLEWARE

Richard E. Schantz, Franklin Webber, Partha Pal, Joseph Loyall
BBN Technologies, 10 Moulton Street, Cambridge, MA 02138, USA
{schantz, fwebber, ppal, jloyall}@bbn.com

Douglas C. Schmidt
Electrical & Computer Engineering Department
Henry Samueli School of Engineering
University of California, Irvine, CA 92697-2625, USA
schmidt@uci.edu

Abstract *A distributed application can be given increased resistance to certain types of malicious behavior, even when the environment in which it is operating contains untrustworthy elements. Recent trends in protecting applications use operating systems as only the first layer of security, anticipating that this layer may be breached. Another layer is added to react to and repair the damage done by intruders that succeed in breaching the first layer. A promising approach to designing the second layer of protection uses adaptive middleware to enable agile behavior and to coordinate protective responses across the distributed system, even in resource-depleted environments. This new approach to protection complements more traditional approaches – in which only one layer of security is used – by hardening critical components at multiple system levels. When integrated effectively, this multi-level approach makes it harder for intruders to corrupt or disable distributed systems and applications.*

This paper presents three contributions to the study of protecting distributed applications against malicious behavior. First, we describe the key ideas and technologies behind the emerging multi-level approach to protecting distributed applications. Second, we explain how these ideas relate to security engineering in general. Finally, we report recent results in evaluating a collection of technologies that implement this approach. These results reinforce the premise that an adaptive middleware approach to increasing survival time and enabling applications to operate through attacks is feasible, though much additional research remains to be done.

Keywords: Distributed systems, Quality of service, Adaptive middleware, Information assurance, Security, Survivability

1. MOTIVATION: COUNTERING MALICIOUS INTRUSIONS IN DISTRIBUTED APPLICATIONS

Distributed applications are growing more interconnected and internetworked. They are also relying increasingly on commercial-off-the-shelf (COTS) hardware and software. As a result, they are becoming dependent on the common information technology (IT) infrastructure used to build distributed applications. The increasing reliance on COTS hardware and software stems from a variety of factors, including deregulation, international economic competition, and time-to-market pressures, as well as the gradual commoditization of the global IT industry.

Unfortunately, the common IT infrastructure contains security flaws, both known and unknown. Computer system intrusions (also known as cyber attacks) that take advantage of these flaws have become ubiquitous. These deliberate and malicious actions against computer systems can result in application corruption, with corrupt applications delivering either faulty service or not delivering any service at all. Disrupting or corrupting critical applications and services is often the specific intent of break-ins and denial-of-service attacks. Vulnerability to such intrusions is a key risk area for open, interconnected automated systems, such as those needed to support e-commerce, as well as those systems that protect critical infrastructure, such as power grids, telecommunications, air transportation, and emergency services. Most of these systems are highly mission-critical and interdependent, i.e., failures in some parts of the infrastructure can have catastrophic consequences for other parts.

It is becoming increasingly clear that we do not yet have a trustworthy computing base that is economical enough to use as the basis for mission-critical distributed applications. Moreover, due to the inherent complexities of developing distributed systems, it is not clear that a completely trustworthy computing base can be achieved in the long run either. A key challenge therefore is how to build applications that exhibit better security and survivability characteristics than the potentially flawed common IT infrastructure they are built upon.

Many intrusions into computer systems target specific applications with an intent to stop them from functioning properly. One way to defend against such threats is to try to ensure the applications' continued ability to provide useful service despite the ongoing attack(s). A new approach to this problem is derived from the concepts of providing a layered defense against intrusions, while continuing to provide varying degrees of service despite the ongoing intrusion. We call applications that can deal with potential flaws in their infrastructure "defense enabled" applications, and we call adding the survivability properties to an application "defense enabling."

The premise of this paper is that it is possible to build more survivable systems using an infrastructure that is not completely reliable and trustworthy. The key to success is the systematic use of *adaptation*, supported by redundancy, heterogeneity, and use of current security mechanisms, such as access-control, intrusion detection and packet filtering. Adaptive behavior is paramount to defense and survival in imperfect infrastructure environments. Since the operating condition of the system can change as a result of even a partially successful attack, the system and application must cope with changes in order to survive.

The need for adaptive behavior is particularly essential for mission-critical distributed real-time and embedded (DRE) applications, such as supervisory control and data acquisition (SCADA) systems used to control power grids. For instance, a cyber attack on a DRE system may consume resources, such as bandwidth, memory, or CPU cycles. A survivable DRE application must therefore be able to either continue with the degraded resources (perhaps also providing degraded levels of service) or actively engage other mechanisms to counter the shortage or degradation. Two things are important in this regard:

1 the adaptation strategy, i.e., what to do in general as a response to various forms of attack or potential attack, and

2 the mechanisms exercised as part of the adaptation strategy, i.e., the mechanisms that act as sensors and actuators for the strategy.

1.1. A NEW GENERATION OF INFORMATION ASSURANCE TECHNOLOGY

The emerging multi-level, adaptive approach toward information assurance is enabling a new generation of techniques and technologies to augment the approaches and technologies of previous generations that have proven to be deficient. One way to categorize the collection of current techniques is through their intent and approach toward contributing to a solution, as shown in Figure 1.

We characterize each generation of information assurance (IA) as follows:

1 First generation IA technologies sought to provide *protection*, i.e., preventing an attacker from getting through a boundary estab-

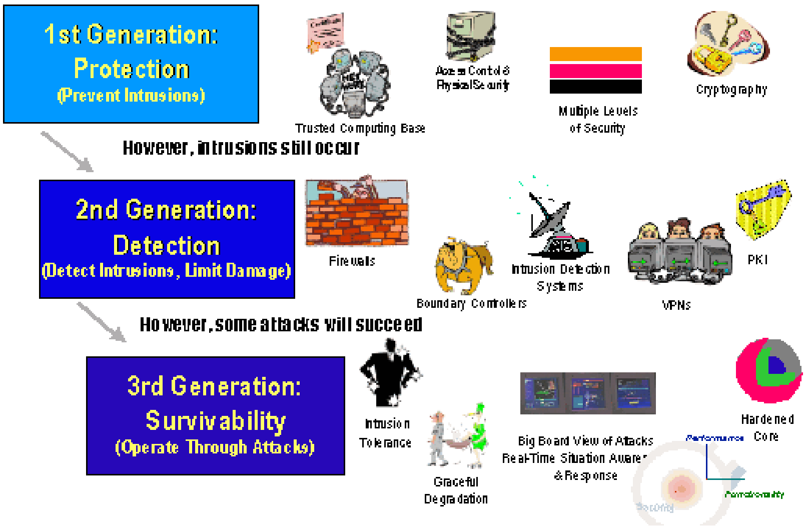

Figure 1 Information Assurance Technology Generations

lished to insulate applications from malicious behavior [1, 29, 27]. Despite significant effort and progress in mechanisms for protection, however, intrusions still occur.

2 Second generation IA technologies focused on intrusion *detection*, attempting to identify successful intrusions and alert system administrators to remedy the situation [9, 15, 10]. Evaluations of intrusion detection approaches were mixed, with some types of intrusions being detected, while others went undetected. Work continues on better intrusion detection, but there is growing realization that neither protection nor detection will be of sufficient quality any time soon to completely resolve the problem.

3 Third generation IA technologies recognize that there will be failures, both in the low-level infrastructure protection and in intrusion detection mechanisms. *Survivability* (or intrusion tolerance) is the goal of these third generation approaches, in particular the ability to operate through the effects of intrusions [16, 14]. Intrusion tolerance has three main tenets: (a) focus and harden protection of key resources, (b) provide layers of protection which complement each other, and (c) use intrusion detection and other means of identifying anomalous behavior to trigger adaptation to cope with the effects of a successful intrusion on limiting the correct functioning of critical applications.

1.2. SURVIVABILITY, THIRD GENERATION INFORMATION ASSURANCE

The premise behind the third generation IA activities is the realization that the number and sophistication of cyber attacks is increasing and that some of these attacks will succeed, despite our current deployment of first and second generation solutions. It is therefore essential that the new generation of distributed systems be capable of *operating through attacks* by:

1 Protecting the most valuable assets,

2 Layering defenses in-depth,

3 Accepting some degradation in service, and

4 Trying to move faster than the intruder.

This approach differs from the static (and often unrealistic) "all or nothing" ideal of previous generations.

Our R&D efforts focus on developing, demonstrating, and deploying a set of technologies that allow any mission-critical distributed software application to resist many forms of malicious intrusions. Our approach is based on the new generation of fine-grained, flexible, late binding adaptive middleware [25] technology that we developed to manage end-to-end application quality of service (QoS) and to enable more agile applications. Our premise is that applications that can adapt to work around the effect of attacks will offer more dependable service, and will minimize the loss from inevitable intrusions. For this approach to be workable, "defense enabling" needs to be applicable without requiring major modifications to the bulk of the application software.

In general, agile responses to survivability are hard since there are so many different avenues to intrusion, i.e., too many to deal with one by one. If an application just reacts to specific intrusions, therefore, it will continuously be playing catch-up with intruders. A more realistic approach is therefore to deal with the symptoms or effects of intrusions, such as excessive bandwidth (or more generally resource) consumption, identifiable resource abuse, identifiable corruption of files and data, loss of QoS delivered to applications, and measuring application progress against some established expectation.

The remainder of this paper discusses the ideas behind the defense enabling concept, as well as some of the current mechanisms and strategies that have been developed, categorized, utilized, and partially evaluated in pursuing it. Section 2 begins by discussing current trends in

QoS management middleware that form a basis for the adaptive security approaches. Section 3 discusses the concept and practice of using the adaptive QoS management technology to defend against intrusions, focusing on the current set of mechanisms we have used and strategies with which we have experimented. Section 4 introduces and evaluates a series of activities we have conducted to ascertain the viability of this new direction. Section 5 presents conclusions drawn from our work to date.

2. CURRENT TRENDS IN MIDDLEWARE: INTEGRATED END-TO-END QOS, LAYERING, AND ADAPTIVE BEHAVIOR

2.1. TECHNICAL BACKGROUND

For a significant number of mission-critical systems, both the current operating environment and information processing requirements can be expected to change during operation. This dynamism often implies the need to adapt to changing conditions. A key challenge is to design a distributed system infrastructure architecture and reify this architecture into a concrete, reusable, multi-level middleware framework for building adaptive applications. The recently completed DARPA Quorum research program has addressed these issues. As depicted in Figure 2, the Quorum architectural framework is organized around QoS-oriented specification, feedback, negotiation, and control mechanisms that can be applied to the wide range of QoS aspects that were investigated in Quorum.

Projects in Quorum developed advanced, reusable middleware that enabled a new generation of flexible distributed real-time and embedded (DRE) applications. These new types of applications can exert more explicit control over their resource management strategies than the previous generation of statically provisioned applications. As a result, DRE applications can be reconfigured and adapted more easily in response to dynamic changes in their network and computing environments. In particular, they have wider operating range than the conventional prevailing binary mode between “working” and “broken”.

¿From one perspective, Quorum projects developed an extensible software development framework, built on a distributed object (DOC) middleware infrastructure that simplifies the support of dynamic run-time adaptation to changing configurations, requirements, or availability of resources. ¿From a complementary perspective, Quorum projects provided the baseline for an evolving network-centric architecture with a growing set of

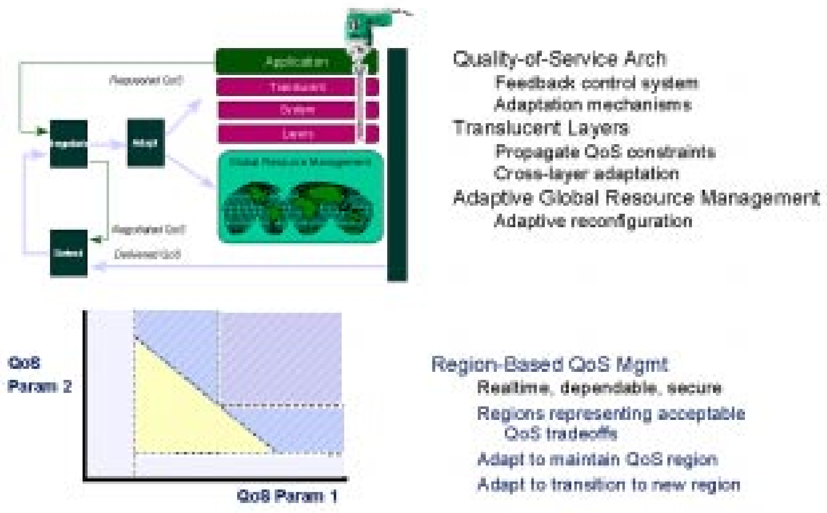

Figure 2 Quorum: Providing Predictable, Controllable, End-to-End Response on a Shared COTS Hardware and Software Infrastructure

- **concepts**, such as contracts to collect and organize QoS requirements, region based QoS resource management, and system condition measurement objects,

- **components**, such as resource status services, real-time event services, and resource managers,

- **Mechanisms**, such as end-to-end priority and reservation resource management, active and passive replication management, dynamic access control,

- **Products**, such as The ACE ORB (TAO) and Eternal, and

- **Standards**, such as those for Real-Time CORBA and Fault-Tolerant CORBA.

Subsequent DARPA programs, such as PCES, MoBIES, and ARMS, are filling out the network-centric architecture to support an integrated QoS concept for managing collections of system aspects and the tradeoffs among these aspects to support a wide range of operating objectives effectively.

2.2. TECHNICAL INTEGRATION IN QUORUM

In any DOC middleware architecture, the *functional path* is the flow of information between a client's invocation to a remote object and back. The middleware is responsible for exchanging this information efficiently, predictably, scalably, and securely between the remote entities by using the capabilities of the underlying network and endsystems. The information itself is largely application-specific and determined solely by the functionality being provided (hence the term "functional path"). The functional path deals with the "what" of the client⇔object interaction from the perspective of the application, e.g., what function is to be requested for that object, what arguments will be provided and what results, if any, will be returned to the client.

In addition to providing middleware that supports the functional path, projects in Quorum added a *system path* (a.k.a., the "QoS path") to handle issues regarding "how well" the functional interactions behave end-to-end. Quorum middleware is therefore also intimately concerned with the *systemic* aspects of distributed and embedded application development, which include the resources committed to client⇔object interaction and possibly subsequent interactions, proper behavior when ideal resources are not available, the level of security needed, and the recovery strategy for detected faults. A significant portion of the Quorum middleware focused on collecting, organizing, and disseminating the information required to manage how well the functional interaction occurs, and to enable the decision making and adaptation needed under changing conditions to support these systemic "how well" QoS aspects.

Quorum projects separated the systemic QoS requirements from the functional requirements for the following reasons:

- To allow for the possibility that these requirements will change independently, e.g., over different resource configurations for the same applications;

- Based on the expectation that the systemic aspects will be developed, configured, and managed by a different set of specialists than those customarily responsible for programming the functional aspects of an application.

In their most useful forms, systemic QoS aspects affect *end-to-end* activities. As a result, they have elements applicable to the network substrate, the platform operating systems, the distributed system services, and the programming run-time system in which they are developed, the applications themselves, as well as the middleware that integrates all these elements together. Thus, the following basic premises underlie adaptive middleware:

- Different levels of service are possible and desirable under different conditions and acceptable cost profiles.
- The level of service in one dimension may need to be coordinated with and/or traded off against the level of service in other dimensions to achieve the intended overall result.

There were three complementary parts to Quorum's middleware organization:

1 The features and components needed to introduce the concepts for predictable and adaptable behavior into the application program development environment, including specification of desired levels of QoS aspects.

2 Providing run-time middleware to ensure appropriate behavior, including the collection of information and coordination of any needed changes in behavior.

3 Inserting the mechanisms for achieving and controlling each particular aspect of QoS that is to be managed, including aggregate allocation, scheduling, and control policies.

Integrating these facets and inserting sample mechanisms and behavior required a significant integration job, which remains an ongoing area of interest in subsequent DARPA programs.

Figure 3 illustrates our general concept of middleware and some of the key layers we are using to organize the technology integration activities. Based on our prototyping and benchmarking activities to date [6, 11], the integrated components enable an unprecedented degree of dynamic application-level control and adaptability to varying conditions typically found in both embedded and Internet environments. In turn, these integrated capabilities enable a new generation of applications whose resource management capabilities can be customized easily, without the need to complicate the task of application developers significantly.

The remainder of this section briefly highlights the various project techniques used to populate the Quorum adaptable system vision to provide the necessary infrastructure components and specific QoS property mechanisms. These activities contribute toward, and form a basis for, the array of mechanisms and adaptations now available for use to develop agile applications that can respond appropriately to successful intrusions. Each of these activities has been described in more detail

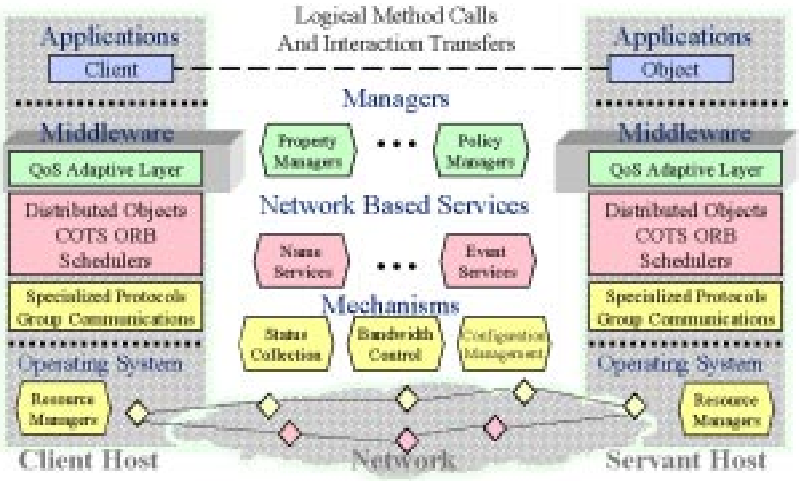

Figure 3 Network-Centric QoS Interface and Control as Part of a Layered Architecture

elsewhere, and we provide pointers to the individual project web sites for additional information on the particular technologies.

Quality Objects (QuO) is a distributed object-based middleware framework developed at BBN [33, 30, 13]. QuO facilitates the creation and integration of distributed and embedded applications that can specify

- Their QoS requirements,
- The system elements that must be monitored and controlled to measure and provide QoS, and
- the behavior for adapting to QoS variations that occur at run-time.

QuO adds the abstractions of a *QoS contract* that summarizes the service requirements of the possible states the system might be in and actions to take when changes occur, *system condition objects* that measure and control behavior, and *delegates* that are packaged inline adaptive behaviors. In addition, QuO introduces the concept of an *Object Gateway* [24] that provides a means to integrate a wide variety of transport-level QoS mechanisms into the DOC middleware paradigm. Figure 4 highlights the components of the QuO framework and their relationship to other parts of the Quorum adaptive QoS environment.

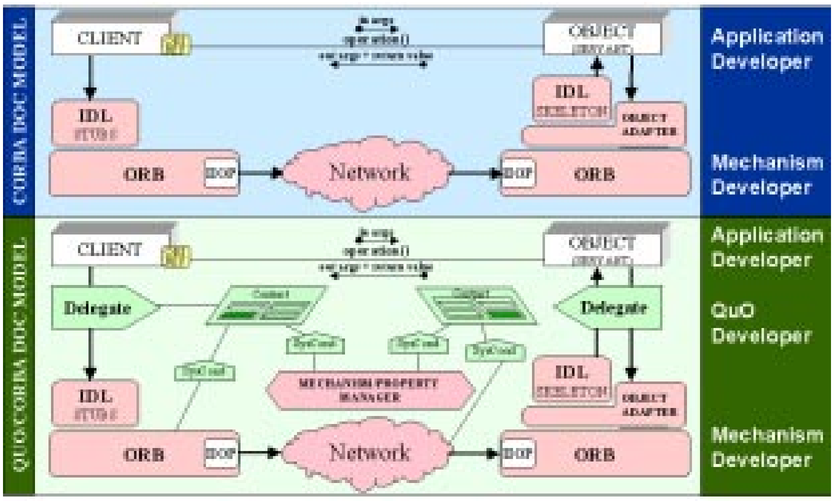

Figure 4 QuO is middleware that offers an application the ability to adapt to a changing environment

The ACE ORB (TAO) is an open-source implementation of CORBA [19] being developed Washington University, St. Louis, the University of California, Irvine, and Vanderbilt University. TAO provides a CORBA compliant, QoS-enabled, COTS middleware object request broker (ORB) and related ORB services. The ORB endsystem encapsulates the network, OS, and communication interfaces and contains CORBA-compliant middleware components and services illustrated in Figure 5.

In addition to supporting the standard OMG CORBA reference model, TAO also supports the Real-time CORBA specification [18], with enhancements to ensure predictable QoS behavior for real-time applications. In particular, TAO provides a real-time object adapter and runtime schedulers for both static and dynamic real-time scheduling strategies [26]. It also provides a real-time event service [7] that enables applications to utilize the Publisher/Subscriber pattern [4] within a CORBA context.

Figure 6 highlights the layering of adaptive QoS middleware over the integrated QuO and TAO real-time CORBA DOC environment in an avionics context. In this view, the task of interfacing to the appli-

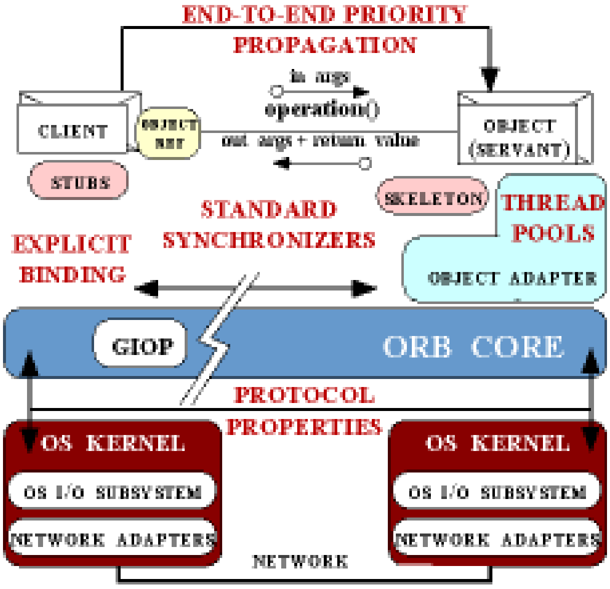

Figure 5 TAO: A Real-Time CORBA-Compliant ORB

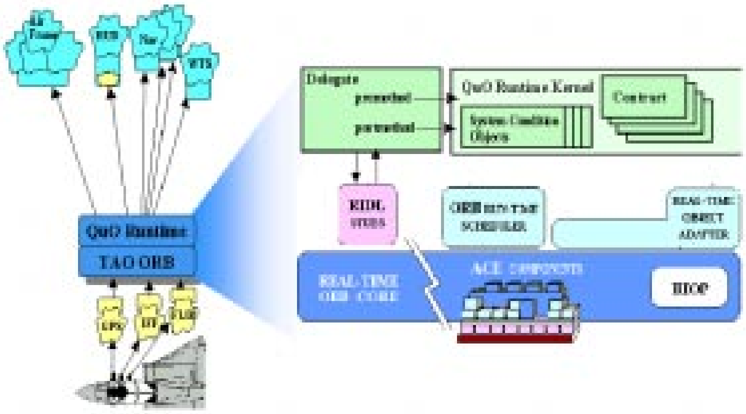

Figure 6 Adaptive Real-Time Behavior

cation is assigned to the adaptive middleware layer, which tracks the progress and changes the control parameters accordingly for the enforcement mechanisms provided by the real-time DOC middleware layer. The adaptive QoS management functions as a higher level of middleware that can interoperate with both the application-specific requirements and the lower-level middleware control mechanisms to produce the desired behavior under varying circumstances. Real-time CORBA mechanisms, such as prioritization and filtering, can be modified dynamically to better match the current application requirements with the current operating conditions, as measured and evaluated by the adaptive middleware layer.

Object-Oriented Domain and Type Enforcement for Access Control. Adaptive security is supported through another Quorum technology called Sigma, being developed by Network Associates Inc. Labs. Sigma is a DOC-based access control mechanism and policy language that employs a domain and type enforcement model. Introducing Sigma into the Quorum environment involved providing adaptive security policies, enforceable through the ORB and the Object Gateway. In addition, Sigma provides a response mechanism that can be connected to a variety of triggers, such as variations in delivered QoS or specific Intrusion Detection Systems (IDS), to form the basis of defensive actions taken under suspicious circumstances. Figure 7 illustrates the concept integrated with DOC. For more details on the adaptive security aspects of this work, see [28, 20].

The **Proteus Dependability Manager** was developed primarily at the University of Illinois based on using off-the-shelf group communication mechanisms (Ensemble [8]) to control the consistency of object replicas. It provides a prototype property (dependability) manager component that coordinates the number and location of object replicas, as well as coordinating the selection of a replica control strategy, from among a growing class of supported strategies with various footprint and fault coverage capabilities. For more details on the design for dependability see [5, 21] and [17] for the emerging CORBA Fault Tolerance standard which this work has influenced. Figure 8 highlights the Proteus design for the dependability aspects in Quorum.

Network Measurement and Control Functions. Measurement and control of network resources are an important part of the bandwidth management QoS capability. The REMOS component from Carnegie Mellon University was developed in Quorum to acquire and disseminate information about network and host resource utilization, and an emerging capability for predicting future use. Integrated Services (IntServ) and Differentiated Services (DiffServ) are emerging Internet standards [32,

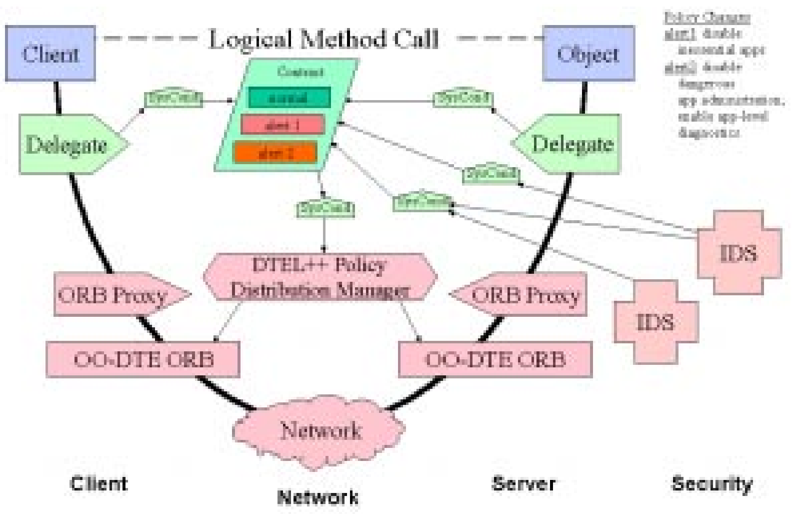

Figure 7 Adaptable Access Control Policy

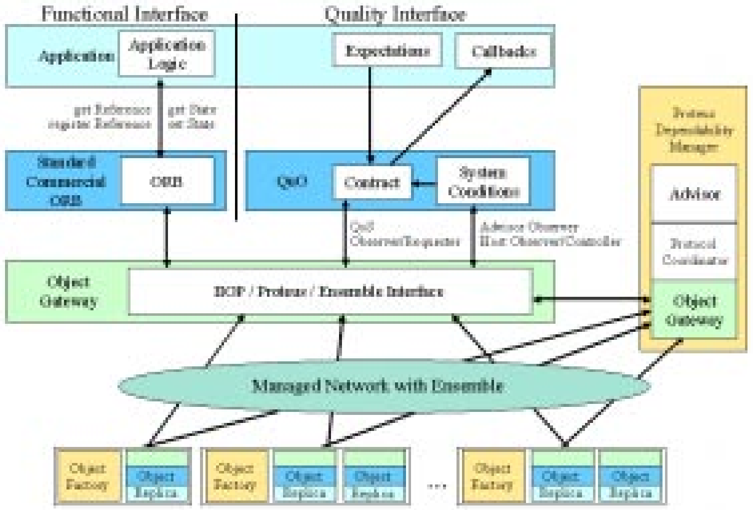

Figure 8 Dependability Using Replication Management

3] for managing network resources, by providing a resource reservation mechanism capability (IntServ) and a priority based mechanisms (DiffServ) for controlling network communication in a more predictable manner. Figure 9 illustrates how these technologies integrate with other components in the Quorum DOC framework.

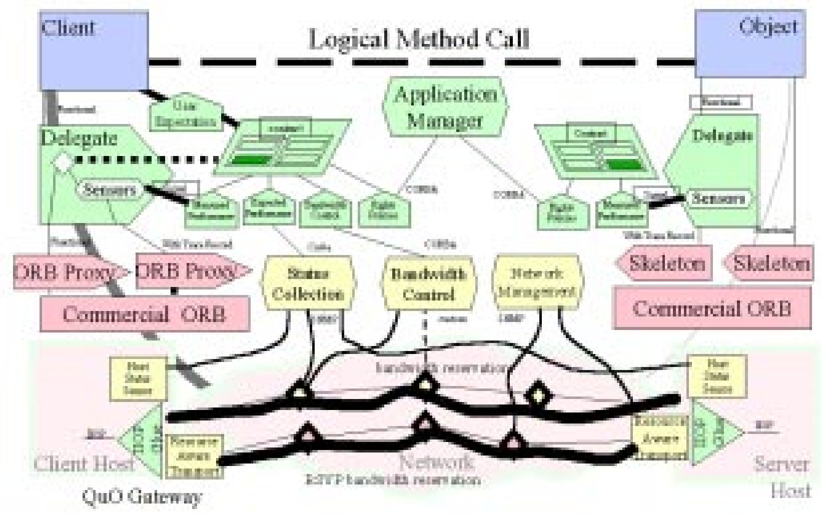

Figure 9 Bandwidth Management Measures and Controls Network Resources

Additional information about the adaptive middleware activities described above can be found at the following websites:

- BBN's QuO website at
  www.dist-systems.bbn.com/tech/QuO/
- the Washington University website for The ACE ORB (TAO) at
  www.cs.wustl.edu/∼schmidt/TAO.html
- The Network Associates SIGMA website at
  www.nai.com/products/security/tis_research/applied/arse_corba.asp
- the University of Illinois AQuA website at
  www.crhc.uiuc.edu/PERFORM/AquA.html
- CMU's Remos website at
  www.cs.cmu.edu/∼cmcl/remulac/remos.html

These Quorum-sponsored activities form the underlying technology basis for a new generation of highly monitored and QoS-managed behavior. In addition, they provide the basis for adapting that QoS-managed behavior to enable survivable applications that can operate through system intrusions. Section 3 outlines how we are applying these underlying technologies in pursuit of a defense enabling strategy using mechanisms based on these controlled and adaptive behaviors emerging from middleware R&D.

3. USING ADAPTIVE MIDDLEWARE TO DEFEND AGAINST INTRUSIONS: CONCEPTS AND PRACTICE

3.1. MOTIVATION FOR DEFENSE ENABLING

Given sufficient time and effort, a determined attacker can in principle defeat whatever flawed protection is offered by operating systems or networks, thus gaining privileges that can be used either to kill the system completely or to corrupt it in some pernicious way. Although one might try to protect data using encryption and digital signatures that are computationally infeasible to break [27], when that data is processed by the system it will almost certainly become vulnerable to an attacker whose intrusion gains enough privilege.[1]

In practice, an attacker may not have the skill, perseverance, preparation, or time needed to carry out the attacks that are possible in the worst case. Some attackers rely on prepackaged attack “scripts” and do not have the skill to repair the scripts if they fail. An attacker who meets unexpected obstacles may look elsewhere for easier targets rather than persevere in an attack. An attacker who is not prepared in advance to circumvent dynamic protection elements in a specific system will be more likely to trigger intrusion detection alarms [9]. In any case, the more time attackers take, the more vulnerable they are to being detected and stopped by system administrators. In summary, system protection is not perfect, but neither are attacks and attackers. Moreover, we can increase the survivability of applications by applying strategies and mechanisms that can limit the effects when protection fails and intruders gain some privileges.

[1] Note that encrypted data is worthless unless it is decrypted at some time, and it can be read at that time by an attacker with sufficient privileges. Also note that digitally signed data must be re-signed when it is modified, and an attacker who gains the privilege to re-sign data can forge new, corrupt data as well.

Flaws in an application's implementation can be corrected more easily than flaws in its environment, and the latter are likely to be better known to attackers and exploited by them. Note that we are assuming we can modify or extend the design and implementation of critical applications. This is in sharp contrast with the design and implementation of the environment, which is largely beyond our immediate control. In other words, we must live with the inevitable flaws in the environment but, because our goal is defending critical applications, we will expend the effort to make those applications much more trustworthy than the operating systems and networks on which they depend.

We make a distinction between *protection*, which seeks to prevent the attacker from gaining privileges, and *defense*, which includes protection but also seeks to frustrate an attacker in case protection fails and the attacker gains some privileges anyway. Protection mechanisms are typically static and proactive; defense mechanisms enhance the protection mechanisms with a dynamic strategy for reacting to partially successful attacks. Both protection and defense aim to keep a system functioning, but protection tends to be all-or-nothing (i.e., either it works or it does not), whereas defense can choose from among a range of responses, some more appropriate and cost-effective than others.

During an intrusion, an attacker and an application both contend for resources, and one key criteria of successful defense is the ability to continue despite changes in the environment caused by the attack. Under these circumstances, putting up a static number of replicas or using a fixed access control policy may not be a successful strategy. The initial deployment, configuration, resource allocation, and service-delivery set-up of an application will need to adapt dynamically. Consequently, requirements generated from the defense strategies often involve recognition of environmental changes and dynamic reaction to these changes.

Under severe resource depletion, for example, certain operations of an application will be made unavailable. Another example involving adaptation involves migration of application objects from a host as soon as an attack on the host is suspected. The strategic adaptations are usually (but need not always be) reactive, i.e., in response to some events. A defense strategy may involve pro-active adaptation as well. For example, to counter the attack on a service, one may include multiple objects providing that service and periodically switch the set of service providers pro-actively.

3.2. DEFENSE ENABLING: MECHANISMS AND STRATEGIES

We say that an application is *defense enabled* if mechanisms are in place that cause most attackers to take significantly longer to corrupt it than would be necessary without the mechanisms. In other words, attackers must not only defeat protection mechanisms in the environment, they must spend additional time defeating defense mechanisms added to the application.

A typical use-case for our technology begins with the defense or survivability needs of a critical application. Based on the attacks or attack effects that this application needs to survive, a defense strategy is then devised. Implementation of this strategy may require identification and integration of external services of defense mechanisms ranging from packet filtering to replication management. The final step of defense enabling is the integration of the defense strategy with the application. We are attempting to develop a catalog of general defense strategies that are independent of the application context and therefore potentially reusable. Implementing the defense in the middleware is a key concept in this regard because of its end-to-end purview. Since middleware is explicitly removed from the application context, yet is close enough to be aware of specific application requirements, it can provide the right combination of integrating the use of diverse infrastructure based mechanisms with application-centric customization.

Defense strategies vary in their objectives. Examples of different strategy objectives include countering the attack, working around the attack by moving elements of the computation, imposing a stronger barrier against future attacks, etc. Various aspects of the system including application-level behavior, QoS management, and organization and configuration of infrastructure resources may be affected and altered by a defense strategy. In theory, any mechanism whose services are useful in implementing a defense strategy can potentially be used as a defense mechanism. In practice, however, one has to overcome the technical challenge of integrating diverse (and often conflicting) mechanisms. For defense, we have used both standard security mechanisms and other mechanisms adapted for security ends. Examples of security mechanisms used as defense mechanisms include access control, packet filtering, and intrusion detection mechanisms. Among the defense mechanisms that are not historically perceived as security mechanisms include replication management and bandwidth management mechanisms.

3.3. IMPLEMENTING DEFENSES USING MIDDLEWARE

It is possible to construct the adaptation and coordination required to implement the strategies at the application level, exercising the capabilities of defense mechanisms directly from application code. This ad-hoc approach is not reusable, however, and controlling infrastructure protection mechanisms are not typically considered an application level prerogative. It also violates the separation of concern between functional and survivability aspects and leads to unwieldy and custom code. A better software engineering practice is to devise and deploy a middleware layer that lies between the application and systems resources to implement the adaptation. Integration of a defense mechanism with the middleware brings the awareness and control closer to the application, while simultaneously decoupling survivability aspects of the application from the functional aspects. We believe that recent advances in adaptive distributed middleware technology (some key elements of which were described in section 2.2) have reached the level of maturity where one can achieve this kind of integration and coordination systematically and without undue difficulty.

3.4. SOME SAMPLE MECHANISMS

Below, we present examples of adaptive middleware mechanisms that can be used to defense-enable applications.

Mechanism: Bandwidth Management Can Counter Flooding Between Routers. Communication bandwidth management mechanisms, such as RSVP, can be used to counter a network flooding attack by reserving bandwidth for a defense-enabled application, as shown in figure 10. RSVP can be given a CORBA interface and "protected" with Access control (OO-DTE/Sigma), but unfortunately RSVP offers other interfaces that an attacker can use to deny bandwidth to the application. Thus, the defense mechanism can be turned to the advantage of the attacker. For this reason, a QoS manager must offer some degree of trustworthiness before being used for defense-enabling. In the case of RSVP, a trusted variant is under development in the ARQoS project [31].

Mechanism: Replication Management Can Replace Killed Components. Replicating key components of an application increases the application's ability to withstand certain types of attacks, e.g., attacks which attempt to make individual components unusable by the rest of the system. The replicas must be coordinated to ensure that, as a group, they will not be corrupted when the attacker succeeds in corrupting some of them. If one replica is compromised, the applica-

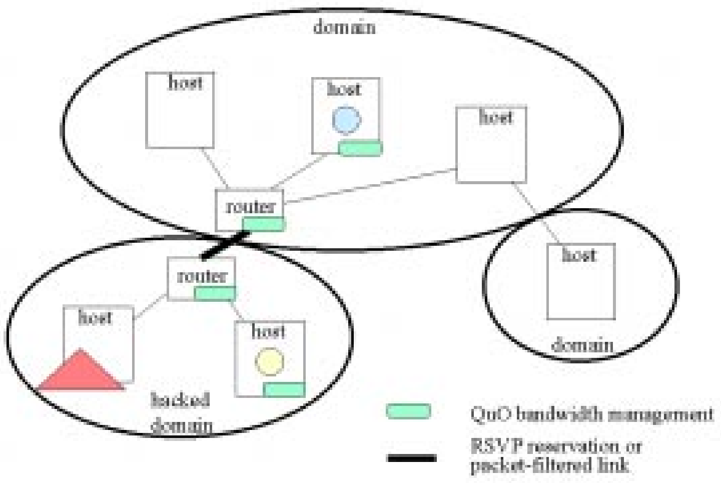

Figure 10 Mechanism: Bandwidth Management Can Counter Flooding Between Routers

tion can rely on the other existing replicas to provide continued service. Replication management systems vary in their ability to tolerate the way replicas can fail (e.g., crash, value, or Byzantine failures). Depending on the underlying failure model, a replication management system may be able to maintain service availability as long as a certain number of replicas have not failed. It may also be possible to dynamically change the replication management strategy. Failure reports obtained from replication management systems can be used to track abnormal failure patterns indicative of a potential intrusion [12].

Some examples of how the services of a typical replication management system may be used in a defense strategy include:

- Increasing replication level to increase the probability of the application's survivability when some hosts are suspected to be compromised by the attacker;

- Migrating replicas from one host when that host is under attack;

- Start voting on responses (instead of accepting the first response) when an attack on some host is suspected (i.e., change replication management strategy).

We have used the replication management facilities of the Proteus dependability manager [22] as a representative example of using replication management as a defense mechanism. We used Proteus' CORBA interfaces to integrate it with QuO system conditions, thereby bringing both awareness and control of the replication management aspects to the adaptive middleware. Figure 11 shows schematically how Proteus can be used as a defense mechanism.

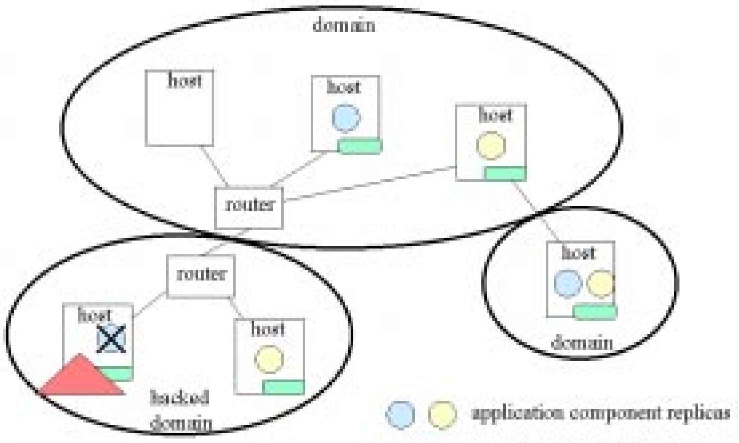

Figure 11 Mechanism: Replication Management Can Replace Killed Components

Mechanism: Security Domains Can Limit the Damage From A Single Intrusion. Defense enabling depends on slowing the spread of privileges to attackers. To see this, note that if privileges could be obtained instantly, the attacker could immediately grab all the privileges needed to stop all application processing and thus deny all service. No defense would be possible against this unlimited attack. To help slow the spread of privileges, we divide the system into several *security domains*, each with its own set of privileges.

A security domain can be a network host, a LAN consisting of several hosts, a router, or some other structure. The domains are chosen and configured to make best use of the existing protection in the environment to limit the spread of privilege. The domains must not overlap. For example, if the domains are sets of hosts then each host is in exactly one domain. Each security domain may offer many different kinds of privilege. The following hierarchy is a minimal set that is typical in many domains:

- **Anonymous user privilege** – allows interaction with servers in a security domain only via network protocols, such as HTTP, that do not require the client to be identified;
- **Domain user privilege** – allows access only to a well defined set of data and processes in one particular security domain (e.g., the user must ”log in” to get this access);
- **Domain administrator privilege** – allows reading and writing of any data and starting and stopping any processing in one particular security domain (e.g., “root” privilege on Unix hosts).

This hierarchy is listed in order of increasing privilege. Each of these privileges subsumes all the previous ones. To increase the protection of critical applications we create a new kind of privilege in each domain:

- **Application-level privilege** – allows one to interact with a defense-enabled application using protocols at the application level (e.g. CORBA calls that query the application or issue commands).

An attacker with application-level privilege would find it easy to control, and thus corrupt, an application, so defense enabling must make it hard for an attacker to get this privilege. Application-level privilege is a key part of defense enabling. It differs from other kinds of privilege in that (a) It is not part of the environment but is created specifically to defend an application (b) It uses cryptographic techniques (c) It does not subsume any of the other kinds of privilege and it is not subsumed by any of them.

The intent of the security domain strategy described above is to force the attacker to take more time accumulating the privileges needed to corrupt the applications. This strategy can succeed if each critical application has parts that are intelligently distributed across many domains so that privilege in a set of several domains is needed to corrupt it, as shown in Figure 12. We assume and attempt to ensure that the attacker cannot accumulate privileges concurrently in any such set of domains.

Mechanism: Using Application Centric Adaptation as a Defense Mechanism. Several components of a defense strategy may involve application-level adaptation. For instance, to cope with an attack that captures the CPU resources of a server, a client may start sending fewer requests to that server, may use cached values instead of issuing a remote invocation, or perhaps begin using a different algorithm or implementation approach which relies less on the resource under attack.

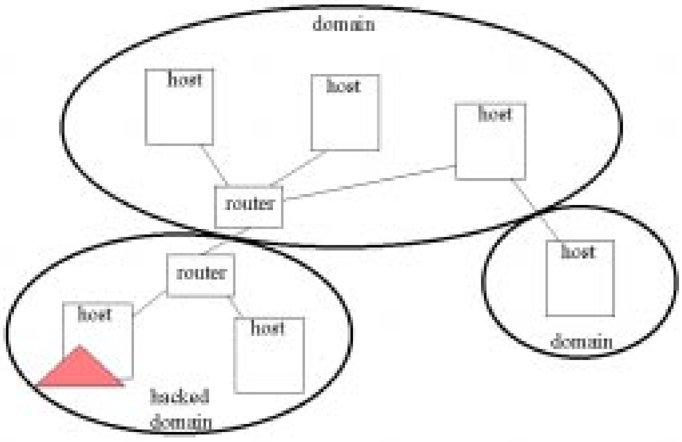

Figure 12 Mechanism: Security Domains Can Limit the Damage From a Single Intrusion

Middleware that supports adaptive behavior can be used for this purpose. The QuO middleware that we use for integrating diverse defense mechanisms provides a systematic way to implement application level adaptation through its contract mechanism.

A sampling of other defensive adaptations we have used and experimented with to isolate or confuse attackers include the following:

- Dynamically configuring firewalls to block traffic
- Dynamically configuring routers to limit traffic
- Dynamically changing communication ports and
- Dynamically changing communication protocols.

3.5. CLASSIFYING DEFENSIVE ADAPTATIONS

An application's defense will use one or more kinds of adaptation to counter a particular attack. This section classifies, in several dimensions, a basic set of potential adaptations, which are shown in Table 1.

The two dimensions shown in Table 1 are described below:

	Defeat Attack	Work Around Attack	Guard Against Future Attack
application level	retry failed request	redirect reqst; degrade srvc	increase self-checking
QoS mgmt level	reserve CPU, bandwidth	migrate replicas	tighten crypto, access control
infrastructure level	block IP sources	change ports, protocols	configure IDSs

Table 1 A Classification of Defense Mechanisms

- In the vertical dimension, adaptations differ according to the level of system architecture at which they work. At the highest level, an application can choose to change its own behavior in the face of an attack, either finding an alternate way to proceed or degrading its service expectations. At the next lower level, the application can use QoS management support to try to make its environment offer the QoS it needs. At the lowest level, the application uses services from the operating system and network level to counter the attack, for example by changing details of how application components communicate

- In the horizontal dimension, adaptations differ according to how aggressively the attack can be countered. At best, the attack can be defeated, i.e., the effect of the attack on the application can be completely canceled. Second best is for the application to work around the attack, avoiding its effects. Finally, if the attack can neither be defeated nor avoided, the application can make changes to protect itself against continuing or similar attacks in the near future.

Although Table 1 shows at least one kind of adaptation for each of the nine possible boxes, the set of adaptations is not intended to be comprehensive. Therefore, others can undoubtedly be invented or would be available with specific operating environments. There may also be other useful categories; for example, Table 1 does not Provide for any "honeypot" defenses, in which an attacker is lured into wasting effort on a decoy. Despite these limits, this set of adaptation mechanisms seems to offer a useful variety of options for creating a strategy for responding to attacks.

A third dimension for classifying adaptations is according to the kinds of attack they work against. In Table 1, the following two broad kinds of attack are countered:

1 **Direct attacks** against the application, for example by disrupting the communication between its parts;

2 **Indirect attacks**, in which resources the application needs are denied.

Direct attacks are countered by the mechanisms working at the application level, plus the use of encryption. An indirect attack might be countered by any of the mechanisms in Table 1 but generally lower-level mechanisms can be more focused. For example, configuring a firewall to block packets from a particular source is a highly focused defense, but one that needs detailed information about the attack to have been collected first. At the QoS level, flooding the network can be countered by bandwidth reservation, over-consumption of CPU can be countered by scheduling and priorities, crashing of a node running an application component can be countered by migrating the component elsewhere, and relatively privileged operations can be disabled with flexible access control if there is a high risk that they might be used maliciously.

A fourth dimension for classifying defenses is whether a mechanism can be used for protection from attack as well as for response to attack, or just for response alone. Mechanisms in Table 1's right-hand column, plus CPU and bandwidth reservation, can be used for protection as well.

It is worthwhile to contemplate why these strategies should not simply always be enabled for best protection? The reason is because some of these defenses, e.g., an IDS configured to be very sensitive to attacks, have significant costs and so may need to be used sparingly. Likewise, others defenses, such as disabling highly privileged operations, impede the normal functioning of the system and so should be used only when necessary. In addition, incorporating many or all of these adaptation mechanisms into a single application can greatly complicate the application's design. Fortunately, every one of these mechanisms is orthogonal to an application's functionality, i.e., the application should compute the same results regardless of whether or how many defense adaptations have been used. In other words, every one of these adaptations changes *how* an application computes its results, not *what* results are computed. This orthogonality allows the design of defenses to be separated from the design of functionality.

As discussed in section 2, the work on separating application functionality from QoS management system behavior, and treating them as separate aspects, feeds directly into the software engineering dimensions of managing defense enabling characteristics as well. It is natural to separate the design of functionality from the design of defenses by putting the latter into middleware [23]. The functionality can be designed first, with a strategy for defensive adaptation added later.

Ideally, the defensive strategy and the mechanisms it uses would be reusable in many different applications, but this is not always possible. For example, access controls are often specific to an application, and self-checking of application invariants will depend on application-specific data structures. These mechanisms seem to be exceptions, however, and most of the other mechanisms in Table 1 can be reused across applications.

3.6. DEFENSE STRATEGIES

We have recently begun to experiment with integration a variety of defense enabling mechanisms into a single, coherent strategy that complement their effects and limitations, and to begin the evaluation and understanding of the survivability effectiveness for these mechanisms working together. Our best current strategy has two parts:

1 **Outrun** the attacker by moving application component replicas off bad hosts and on to good ones faster than they can be compromised. This approach is currently performed using pure replacement only (a policy decision). In the future, we will implement other options, such as increasing the number of replicas or reducing the functionality.

2 **Contain** the attacker by quarantining bad hosts and bad LANs, limiting or blocking network traffic from them and, within limits, shutting them down so they do no further damage. Containment is triggered by intrusion detectors, flooding, or other detected anomalous behavior.

We use the QuO middleware to coordinate the collection of selected defense mechanisms within the selected strategy. Elements from this strategy have been and are undergoing rigorous evaluation, as described in Section 4.

4. VALIDATION AND EXPERIMENTATION

The defense-enabling approach can be validated in several ways, each of which will be described in detail in this section:

- Modeling, which constructs a mathematical object that encapsulates key features of the system, then reasons about the properties of this object.

- Testing, which ensures that individual defense mechanisms work as expected.

- Intrusion injection, which creates situations that might arise during an attack, but which the designers believe are impossible or very hard to arrange.

- Red Team experiments, which seeks attacks that are not foreseen by the original system designers.

We are applying each of these validation techniques to defense-enabled distributed applications to learn whether defense enabling increases resistance to attack, by how much, and which parts of the defense are weakest. In the process, we are also learning which of these techniques gives the best insights into defense enabling and the most accurate measurements of its value.

4.1. VALIDATION BY MODELING

This approach involves constructing a mathematical object that encapsulates key features of the system and then reasoning about the properties of this object. The model must include some features of the application being defended, the defense mechanisms and defense strategy, and assumptions about what the attacker can and cannot do. In general, we do not know for certain what the attacker cannot do, so the model's assumptions about the attacker are necessarily statements about attacks that are unlikely or hard to carry out. Whenever possible, we base such assumptions on observation of real-world attacks.

To date, we have constructed models of replication management and of attack containment using dynamic firewalls. Reasoning about these models allows conclusions such as:

- Conditions under which one management strategy is better than another;

- Estimates of how much extra survival is gained using a particular strategy;

- Requirements on the environment, such as the quality of intrusion detectors, needed to gain a specified level of survivability.

The other validation methods differ from modeling in that they work with the actual defense-enabled system at a software code level rather than an abstraction of it.

4.2. VALIDATION BY TESTING

This approach involves ensuring that the individual defense mechanisms work as expected. This kind of validation is, of course, part of the normal software development process. It is specialized for defense enabling by testing the effects of expected kinds of attack. For example, when using replication management, one expects that the attacker will be able to kill some replicas. One tests the mechanism under this attack to ensure that the mechanism responds correctly, but also to measure the response time and overhead; these measurements will be used in the validation by modeling already described.

4.3. VALIDATION BY INTRUSION INJECTION

This approach goes beyond ordinary testing by creating situations that might arise during an attack, but which the designers believe are impossible or very hard to arrange. For example, a replica coordination protocol may be designed to reject any message that comes from a replica known to have behaved incorrectly in the past and therefore likely to have been damaged by the attacker. Intrusion injection would be used to "inject" such messages at a point in the protocol where they should not exist and thus test what would happen in this thought to be impossible situation. To date, we only started to experiment with intrusion injection.

4.4. VALIDATION BY RED TEAM EXPERIMENT

This approach goes beyond ordinary testing by seeking attacks not foreseen by the designers. Experience shows that a system's designers (also called the Blue Team) are not the best people to find new kinds of attacks against the system they designed. Needed is a different group of people, (the Red Team), who will find it easier to think about the system in new ways and who are skilled in finding system vulnerabilities.

A Red Team experiment is a test in which attacks on the system are tried and the results observed. The Red Team is given

- The system, including defense-enabled applications;
- A set of goals, called "flags" (e.g., "cause the system to deny service for 15 minutes")
- A set of constraints, also called "rules of engagement", that prohibit the Red Team from attacking known vulnerabilities that

could be closed in known ways (e.g., "killing client C is prohibited because in a real system it will be protected by some other mechanism not reproduced in the experiment testbed").

The Red Team then tries to "capture the flag" without going "out of bounds", i.e., without violating the rules of engagement.

We have run two Red Team experiments against a defense-enabled application. The Red Team was provided by an independent set of experts from Sandia National Laboratory (USA) under separate funding by DARPA. The application was a simple video image server, shown in Figure 13, in which clients use a broker to locate an appropriate image server, then get images directly from that server. The defenses included replicating the broker (as shown), and other mechanisms already described in this paper. The primary flag was denying access to the broker replicas.

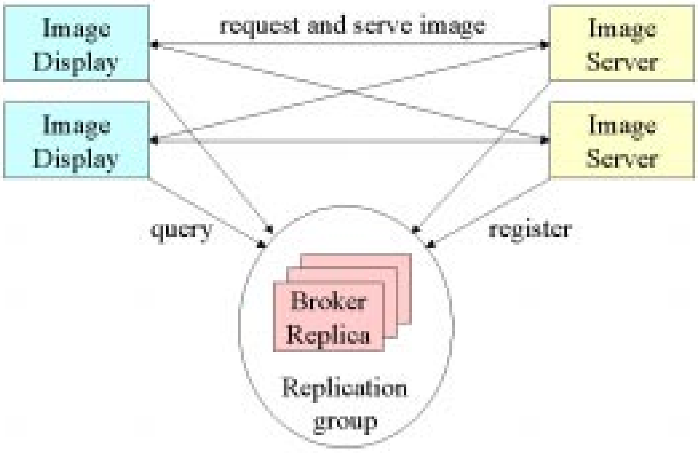

Figure 13 Simplified Application Used in Red Team Experiments

The defense-enabled application was run in the testbed shown in Figure 14. This testbed provided 14 hosts on 4 LANs for application processes, plus a number of other hosts needed for routing and control of the network interconnections.

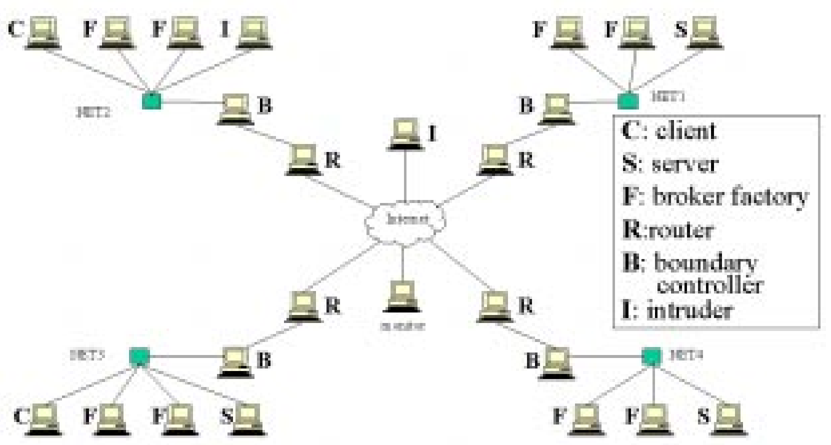

Figure 14 Experiment Testbed

The Red Team used combinations of the following basic attacks against the application, starting out with "root" (i.e., system administrator on Unix) privilege "given" to them on a single host:

- Spoofed scans to cause the defense to (mistakenly) quarantine good hosts;
- ARP cache poisoning to isolate hosts or partition the network;
- TCP connection floods to consume ports;
- TCP connection resets and bad traffic to disrupt communication;
- Network flooding to delay or deny service;
- Replay of RSVP traffic or injection of bogus traffic to stress the bandwidth reservation mechanism.

We observed the following outcomes from both Red Team experiments.

- The Red Team spent significant amounts of time (measured in weeks) learning about all the defense mechanisms and significant amounts of time (measured in days) discovering the attacks that captured the flag.

- After the attacks were automated in scripts, the shortest times to capture the flag were 5-10 minutes. This represents a baseline against which to measure continuing improvements in defense strategies and mechanisms.
- Attacks that captured the flag always set off numerous alarms soon after the start of the attack and before service was denied.
- The Red Team was always eventually able to capture the flag (i.e., deny service), given enough time and gathered experience.

Not all perspectives on red team attacks were exercised in these initial experiments:

- Stealthy attacks were not devised and attempted. So this group of experiments did not yet inform us about the difficulty of capturing the flag without setting off warning alarms.
- Attacks to date did not include attempts to gain privilege on hosts other than the one on which the Red Team was given "root" privilege. It was felt that such attacks would mainly tell us whether system administrators had applied the most recent operating system patches and not whether defense enabling increased resistance to attack. This means that this version of Red Team experiments did not yet validate the degree to which our replication management defense would be able, in practice, to "outrun" an attacker, i.e., start replacement replicas faster than they can be killed.
- Although it was known in advance that an attack that corrupted the in-memory image of any application process could be used to capture the flag, the Red Team decided not to attempt this attack. Such an attack would work because in the current implementation we used protocols that are only crash fault-tolerant and would therefore not tolerate faults that are arbitrarily malicious or Byzantine [2]. Since the application and key parts of the defense were written in Java, and because Java moves data structures around in memory, the Red Team judged that other kinds of attack would yield greater effect for less effort. This means that these Red Team experiments do not yet provide information about the cost of preparing an attack that would motivate Byzantine fault-tolerant protocols to defend against.

Continuing experiments are being planned to fill these gaps in learning about and evaluating the survivability of the current implementation.

We draw the following conclusions from the observations based on our experiments:

- Defense enabling will force even highly skilled attackers to work hard to deny Service. The best attacks can be automated and therefore eventually can be applied by people with fewer skills.

- A defense-enabled application increases survival time (because an attacker with "root" can kill a non-defense-enabled application immediately, so 5-10 minutes is a significant increase and represents a baseline against which to measure progress), but we expect that survival times of 20-30 minutes will be needed as a practical matter to give human operators time to intervene in an attack.

So our current implementation of defense enabling has survival value, but may not yet be good enough for practical purposes.

Our Red Team experiments to date have not yet settled a key question about the value of defense enabling: *In a race between ever-cleverer attacks and ever-improving defense, which side tends to win?* If the attack were to win, survival times would likely shrink to near zero. If the defense were to win, survival times would likely grow to 20-30 minutes or longer. Our series of two Red Team experiments was not long enough to provide an answer. Between the experiments, we added new mechanisms (e.g., a defense against the TCP connection flood used in the first experiment) and we inadvertently introduced new flaws, some of which were exploited by the Red Team in the second experiment. Only a longer series of experiments can determine whether the evolving defense would converge to a stable set of mechanisms, comprehensive enough to block all quick Red Team attacks and trustworthy enough not to be a vulnerability itself.

We argue on general grounds that the mechanisms of defense enabling are simple enough that a trustworthy implementation of them is practical. But we have not yet demonstrated such an implementation. Certainly the defenses we have demonstrated are conceptually simpler than most general-purpose operating systems. Therefore, creating trustworthy adaptive defenses should be easier than the problem of creating trustworthy operating system protection.

5. CONCLUDING REMARKS

A new approach to computer security is emerging, with the following features:

- Adaptive defenses are used to complement the more traditional protection that operating systems and networks are expected to provide.

- Adaptive defenses are organized in middleware, with variations in QoS used as a key indicator of intrusions and control of QoS used as a key component of intrusion response.

- Since defenses adapt to the effects of an intrusion, rather than according to a diagnosis of their cause, the mechanisms used for defense are similar to those already built for reliability, availability, bandwidth, real-time, and other QoS aspects.

We have built adaptive software defenses of the kind described in this paper. That experience leads us to the following conclusions about the value of such defenses and about the software engineering that should support them:

- *The effectiveness of the defense depends crucially on the policy that governs adaptation.* Adaptation polices must answer questions such as "Where and when should new application component replicas be started?", "Should the choice be made in a way that's unpredictable for the attacker?", "Should a decision to quarantine a seemingly corrupt host be made locally or globally?", and "What kinds of evidence are sufficient to decide that a host is corrupt and must be quarantined?".

- *More validation of defense enabling is needed.* Some adaptive defenses have already been evaluated in Red Team experiments. The experiments show that these defenses do increase the resistance of critical applications to malicious attack, but they also show that the defenses themselves are not yet robust enough for practical use.

- *Changing requirements and unanticipated conditions must be handled as part of dependable computing.* If we are ever to truly depend on the mission-critical applications, including e-commerce, air transportation, and power grid control, these systems must be well behaved, even when all of the best case design conditions do not hold. In many cases, introducing major interruptions in service, including self-denial of service when we shut our systems down or take them offline at any sign of trouble, is not an adequate response.

- *Operating with less than a full complement of resources is feasible if the existing resources are used to support the critical application components.* Our experimentation has demonstrated that it is feasible and practical for applications to continue to operate effectively with less than the optimal required resources. However, our results are as yet rather application-specific. The big challenges ahead in this regard are making these approaches more standardized, off the shelf, and cost effective.

- *Late binding distributed object middleware is an avenue to many innovative approaches.* Without late binding capabilities, we cannot hope to achieve the flexibility we need for agile, reactive system behavior. On the other hand, we need to continue to drive down the cost of these late binding mechanisms so that we can still get adequate performance when we use them. We've only scratched the surface on late binding decisions and policies to drive these bindings.

- *Layered solutions with integrated parts are an important development tool, especially for large, complex problems. This involves information sharing and cooperative behavior across and between these layers.* Our systems have grown to cover very large problem areas and scope. One important technique in our ability to field such systems Is to develop the systems in layers and parts, which are often both separately developed or acquired, and later embedded in larger systems. As we try to control the growing end-to-end behavior of these systems, we must of necessity control the behavior of each of the parts in light of the goals for the system as a whole. This is a continuing challenge which must be met, unless we are willing to redo our entire COTS infrastructure or discover new ways to conceive of systems in larger chunks than we now comfortably and reliably know how to build.

The overall challenge going forward is to provide a proper foundation for building our increasingly expansive critical systems in a way which not only combines all of requirements for high performance, realtime connectivity with the physical universe, in a truly dependable manner, but also uses cost effective COTS solutions and cost effective software engineering practices which are both repeatable and easy to use by the people responsible for the development. Making applications responsive to their changing environment and applying this adaptive behavior to survivability and defense behavior is an important new direction. Adaptive middleware provides the basis for augmenting and complementing more traditional operating system and network based protection.

Acknowledgments

A number of other organizations and individuals have made significant contributions, directly or indirectly, to the activities we reported in this paper. We gratefully acknowledge the contributions of Michael Atighetchi, Chris Jones and Paul Rubel from the BBN Distributed Systems Technology group, Bill Sanders, Michel Cukier and colleagues from the University of Illinois reliability group, and all of the developers who contributed to the middleware base of QuO, TAO, AQuA, Remos and Sigma derived from DARPA's Quorum program. In addition, the results reported here benefited greatly from contributions of technical teams from North Carolina State University in providing a secure RSVP implementation, and Sandia Laboratories in performing the red-team evaluations of the current prototypes. Finally, we'd like to acknowledge the various program managers from the U.S. Defense Advanced Research Projects Agency, which funded both the development and the initial evaluations of these new ideas and directions for adaptive systems and information assurance.

References

[1] James P. Anderson. Computer security technology planning study. Technical Report ESD-TR-73-51, vols I and II, AD-758 206, USAF Electronic Systems Division, October 1972.

[2] M. Barborak, M. Malek, and A. Dahbura. The consensus problem in fault-tolerant computing. *ACM Comp. Surv.*, 25(2), 1993.

[3] S. Blake et al. An architecture for differentiated services. Technical Report RFC 2475, Internet Engineering Task Force, http://ietf.org/rfc/, December 1998.

[4] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture – A System of Patterns.* Wiley and Sons, New York, 1996.

[5] Michel Cukier et al. AQuA: An adaptive architecture that provides dependable distributed objects. In *IEEE Symp. Reliable Distributed Syst.*, pages 245–253, October 1998.

[6] B.S. Doerr et al. Adaptive scheduling for real-time embedded information systems. In *Proc. 18th IEEE/AIAA Digital Avionics Syst. Conf.*, 1999.

[7] Tim Harrison, David Levine, and Douglas Schmidt. The design and performance of a real-time (CORBA) event service. In *ACM Conf. Object-Oriented Prog., Syst., Lang., and Applications*, October 1997.

[8] Mark Hayden. The Ensemble system. Technical Report 1662, Cornell Univ., January 1998.

[9] S. Kent. On the trail of intrusions into information systems. *IEEE Spectrum*, December 2000.

[10] Gene Kim and Eugene Spafford. The design and implementation of Tripwire: A filesystem integrity checker. In *Proc. 2nd ACM Conf. Computer and Communications Security*, 1994.

[11] J.L. Loyall et al. Comparing and contrasting adaptive middleware support in wide-area and embedded distributed object applications. In *Int'l Conf. Distributed Comp. Syst.*, April 2001.

[12] Joseph P. Loyall et al. Building adaptive and agile applications using intrusion detection and response. In *Proc. ISOC Network and Distributed Systems Security Conf.*, February 2000.

[13] J.P. Loyall, R.E. Schantz, J.A. Zinky, and D.E. Bakken. Specifying and measuring quality of service in distributed object systems. In *IEEE Int'l Symp. Object-Oriented Real-Time Distributed Comp.*, April 1998. Kyoto, Japan.

[14] Malicious- and accidental-fault tolerance for internet applications. http://maftia.org. IST Programme RTD Research Project IST-1999-11583.

[15] Peter G. Neumann and Phillip A. Porras. Experience with EMERALD to date. In *Proc. 1st Usenix Workshop on Intrusion Detection and Network Monitoring*, April 1999.

[16] Organically assured and survivable information systems. http://www.tolerantsystems.org. DARPA.

[17] Object Management Group. *Fault-Tolerant CORBA Using Entity Redundancy RFP*, 1998. OMG document orbos/98-04-01.

[18] Object Management Group. *Real-Time CORBA Joint Revised Submission*, 1999. OMG document orbos/99-02-12.

[19] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 2.6 edition, December 2001.

[20] Partha P. Pal et al. Open implementation toolkit for building survivable applications. In *DARPA Info. Survivability Conf. and Expo.*, January 2000.

[21] Y. (J.) Ren et al. Passive replication schemes in aqua. In *Proc. Pacific Rim Int'l Symp. Dependable Computing (PRDC)*, December 2002. Tsukuba, Japan.

[22] C. Sabnis et al. Proteus: A flexible infrastructure to implement adaptive fault tolerance in AQuA. In *Proc. 7th IFIP Working Conf. on Dependable Computing for Critical Applications*, pages 137–156, January 1999.

[23] R. Schantz and D. Schmidt. Research advances in middleware for distributed systems. In *World Computer Congress*, August 2002.

[24] Richard E. Schantz et al. An object-level gateway supporting integrated-property quality of service. In *IEEE Int'l Symp. Object-Oriented Real-Time Distributed Comp.*, May 1999.

[25] Richard E. Schantz and Douglas C. Schmidt. Middleware for Distributed Systems: Evolving the Common Structure for Network-centric Applications. In John Marciniak and George Telecki, editors, *Encyclopedia of Software Engineering.* Wiley & Sons, New York, 2001.

[26] D.C. Schmidt et al. Software architectures for reducing priority inversion and non-determinism in real-time object request brokers. *J. Real-Time Syst.*, 2000.

[27] B. Schneier. *Applied Cryptography.* John Wiley & Sons, 1996.

[28] Daniel Sterne et al. Scalable access control for distributed object systems. In *8th Usenix Security Symposium*, August 1999.

[29] US Department of Defense. *Trusted Computer System Evaluation Criteria (Orange Book)*, December 1985. DoD 5200.28-STD.

[30] Rodrigo Vanegas et al. QuO's runtime support for quality of service in distributed objects. *Proc. Middleware 98, the IFIP Int'l Conf. on Distributed Systems Platform and Open Distributed Processing*, September 1998.

[31] T-L. Wu et al. Securing QoS: Threats to RSVP messages and their countermeasures. In *Int'l Workshop on Quality of Service*, June 1999.

[32] Lixia Zhang et al. RSVP: A new resource ReSerVation protocol. *IEEE Network*, September 1993.

[33] John A. Zinky, David E. Bakken, and Richard E. Schantz. Architectural support for quality of service for CORBA objects. *Theory and Practice of Object Systems*, 1(3):55–73, April 1997.