Middleware Policies for Intrusion Tolerance: A Position Statement for WDMS ’02

Franklin Webber, Partha Pal, Chris Jones, Michael Atighetchi, Paul Rubel
BBN Technologies

For many years, researchers have argued that redundancy can be used to recover from computer system failures, not only those failures that arise from random “acts of God” but those caused by malicious and orchestrated acts of Man as well[2]. Some middleware has been built to coordinate groups of component replicas in a way that tolerates arbitrary failures of a subset of the replicas[6][4]; then if an attacker corrupts only one of these subsets, the system will continue functioning correctly. Ongoing research seeks to refine this approach to *intrusion tolerance*, including building the next generation of dependable middleware to support it[5][1].

Our position is that successful intrusion tolerance will depend on a policy that links replica coordination with other intrusion countermeasures and that this policy should be implemented in middleware. Replica coordination by itself is not sufficient because:

- Attackers will try, and often succeed, in corrupting or killing more replicas than can be tolerated. A policy for replacing corrupt or dead replicas automatically is therefore needed to increase the system’s useful life.

- Attackers will try to kill many replicas at once, or corrupt replicas so that each behaves normally at first, then many fail simultaneously. A policy that uses other mechanisms, intrusion detectors and firewalls in particular, to quarantine the attacker is therefore needed to make this and other attacks harder.

The intrusion tolerance policy belongs in middleware because

- it is likely to be reusable for many different distributed applications, and

- it involves adaptation and reconfiguration that needs to be coordinated across multiple hosts.

Intrusion tolerance policies involve a trade-off: a system that is quick to replace replicas and to quarantine hosts where suspicious events have happened is a system that may make itself especially vulnerable to denial-of-service attacks. An attacker who learns to trigger quarantining, for example, may be able to quarantine so many hosts that the system fails to provide enough resources even for authorized applications. On the other hand, a system that is slow to react may be too slow to counter many attacks.

We have defined several intrusion tolerance policies and implemented them using the QuO adaptive middleware toolkit[3]. One of these policies is currently being evaluated in an adversarial “Red Team” experiment. Section 1 describes that policy. Section 2 summarizes the result of a Red Team experiment involving middleware implementing that policy. Finally, section 3 lists some unanswered questions that may lead to further discussion at the Workshop.

1 An Example Policy for Intrusion Tolerance

This section describes a policy for defending a distributed application against intrusions. The policy coordinates the following mechanisms:

- **replica management**: support for detecting replica crashes and starting new replicas;

- **packet filtering**: firewalls, one per host, that allow network traffic to be blocked according to rules that can be established and changed at run-time;

- **intrusion detection**: detectors on every host that look for suspicious network traffic and improper file system activity;

- **access control**: cryptographic support for detecting and rejecting unauthorized modification to application executables and to application- and middleware-level communication.

The policy has the following clauses:

1. Use access control to ensure that every replica gets run from a legitimate executable and that all application and middleware communication is valid.

2. Maintain two replicas of every application component, starting a new one whenever necessary, and coordinate the replicas to tolerate a crash of either replica or a network partition that makes one replica inaccessible.

3. If improper file system activity is detected on host H, mark H as "suspect". If shutting H down would leave at least half of the original set of hosts up, then shut H down and block all network traffic to and from H.

4. If suspicious network traffic is detected from host H, mark H as "suspect". If H is not a host on which application components can be run, block all network traffic from H; otherwise, if shutting H down would leave at least half of the original set of hosts up, then shut H down and block all network traffic to and from H.

5. When starting a new replica, place it, if possible, on a host that is not "suspect".

This intrusion tolerance policy is completely automatic: the response to an attack is completely coordinated by middleware, without any human intervention. Automating the policy means that even a very fast attack may face an effective response. Slower attacks would allow time for human intervention, possibly resulting in a better intrusion tolerance, but such intervention is not part of the policy.

Three issues about this policy must be noted:

1. Arbitrary, i.e., Byzantine, corruption of replicas can cause an application to fail. This policy maintains too few replicas to tolerate an arbitrary failure of one of them, and the replica coordination described in policy clause 2 tolerates only crash failures. However, the intent of policy clause 1 is to make it difficult for even a privileged attacker to corrupt a replica. Tolerating only crash failures is adequate, therefore, to counter most attacks.

   We plan to use Byzantine fault-tolerant protocols[1] in the future, and the intrusion tolerance policy will need to be modified accordingly.

2. Quarantining a host, in policy clauses 3 and 4, involves both shutting that host down and refusing to talk to it. Simply shutting it down may not be sufficient because a privileged attacker on that host may disable the shutdown mechanism before it can be activated.

3. The middleware must synchronize the shutdown of "suspect" hosts so that no more than half the hosts will be shut down unless the attacker has actually infiltrated more than half of them (in which case the attacker could shut them down himself). Policy clause 4 allows for the possibility that the attacker might set off intrusion detectors on many or all of the hosts simultaneously, without having gained privileges on all the hosts. In that case, the middleware must choose which "suspect" hosts to shut down and which to keep, and this choice must be consistent across the network. The constraint that no more than half the hosts should be shut down is an arbitrary limit (e.g., one third or two thirds might be chosen instead) that prevents the middleware defenses from causing a denial of service.

We have implemented this policy in CORBA-based middleware, entirely in Java. Our implementation runs on Linux and uses IPTables for packet filtering and Snort and Tripwire as intrusion detectors. Our access control and replica management are home-grown, but the former uses the cryptographic libraries available in Sun's Java Cryptography Extension (JCE).

2 Results From A Red Team Experiment

An experiment has been conducted to test the effectiveness of our implementation of the policy described in the previous section. This experiment consisted of repeated attacks against our software by a professional "Red Team" from Sandia National Laboratory. This section describes the result of that experiment.

As a context for the experiment, we implemented a simple video display application, in which servers feed image data to clients, and clients periodically select an appropriate server using an intermediate "broker" component. The application is distributed over a network of 13 hosts on 4 separate LANs. It is implemented entirely in Java and uses CORBA.

We defended this application using the intrusion-tolerant middleware previously described. The goal of the defense was to ensure that at least one client continue to receive video images from some server for as long as possible. To concentrate the attacker's attention on our defenses, we protected only the broker component and put the clients and servers off-limits for the attacker. (Clients and servers might also have been protected by our middleware, or they might be fixed resources protected by some other means.) Thus the broker was replicated, and broker replicas might be created anywhere on the network. The goal of the attack was to disable the broker component, thus preventing any client from locating any server and thus preventing the receipt of video images.

The Red Team was given "root" privilege on one host on one of the 4 LANs. The Red Team then spent roughly a week of effort repeatedly attacking the system before finding a way to defeat our defense. That attack method was then automated. The completely automated attack takes approximately 5 minutes to defeat our defense and immediately sets off numerous alarms.

We consider this experiment result a success for our middleware defense. It shows:

- An unprepared attacker, even with a lot of skill and some "insider" privilege, would be forced to spend significant effort to find a successful attack.
- A prepared attacker with some "insider" privilege would likely be forced to spend minutes, rather than seconds, to disable an application.

The successful attack combined three separate techniques to defeat our defense:

1. ARP cache poisoning on the LAN on which the attacker has privilege, to create an artificial network partition and isolate replicas on that LAN;
2. spoofed port scans that seemingly come from hosts on the other LANs, causing our defense to shut down some hosts on which replicas run;
3. TCP connection flooding to block the mechanism for starting new replicas.

So the successful attack disabled the broker component by isolating some replicas, causing the defense to kill others, and preventing new ones from being started.

A follow-on experiment is being prepared. This second experiment will test improved defenses that would have blocked the attack that succeeded in the first experiment. It will also exercise some techniques the Red Team did not explore in the first experiment. In particular, the Red Team will try to cause malicious corruption in at least one replica, thus circumventing clause 1 of our policy, and we will measure the difficulty of doing this.

3 Middleware Issues

One key issue that remains unresolved in our work is the value of unpredictability for defense. Does a defense that chooses unpredictably between alternative countermeasures to an attack make the attacker work significantly harder? For example, the policy in section 1 might start new replicas on randomly-chosen available hosts or it might always choose the available host with the smallest IP address. Intuitively it seems the attacker must work at least as hard to defeat the unpredictable defense, but is the increase in difficulty significant in practice?

This issue can be directly related to middleware because distributed middleware can be a source of unpredictability itself. Nondeterminism in a distributed computation can lead to unpredictable choices between alternatives. Can this nondeterminism be harnessed to improve intrusion tolerance?

A second, related issue is diversity. Our approach to intrusion tolerance depends on diversity to work: without diversity, if all replicas run the same code, run on identical platforms, and run in environments that are administered identically, then an attack that kills one replica should work to kill them all at once. With diversity, multiple implementations, multiple platforms, and heterogeneous environments should increase intrusion tolerance.

Clearly, middleware is an important way to manage diversity, allowing an application to be given a consistent intrusion tolerance policy that spans multiple, heterogeneous platforms. But how much diversity is necessary for intrusion tolerance, and does the need for more diversity add any new middleware requirements?

A third issue is packaging intrusion tolerance policies, such as the one in section 1, for reuse across different applications. Clearly, encapsulating the policy in middleware helps reuse. However, we have little experience so far in applying the same policy to radically different applications. Can the policy be easily parameterized by relevant factors such as the number and kind of hosts, and the network topology?

A fourth issue so far unaddressed by our work is the relationship between real-time constraints and intrusion tolerance. The addition of real-time constraints may make a distributed computation more fragile and thus easier for an attacker to disrupt. On the other hand, small disruptions that result from an intrusion may be more easily detected in a real-time system and thus may be countered sooner. Whether real-time applications are harder or easier to make intrusion-tolerant is an open question.

References

[1] Intrusion tolerance by unpredictable adaptation. http://itua.bbn.com. BBN Technologies and University of Illinois.

[2] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Trans. Prog. Lang. Syst.*, 4(3):382–401, 1982.

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

[4] L. E. Moser et al. The Eternal system. In *ACM Conf. Object-Oriented Prog., Syst., Lang., and Applications*, Oct. 1997.

[5] D. Powell et al. MAFTIA (malicious- and accidental-fault tolerance for internet applications. In *Int'l Conf. Dependable Syst. and Networks*, July 2001.

[6] M. K. Reiter. Distributing trust with the Rampart toolkit. *Commun. ACM*, 39(4):71–74, Apr. 1996.