Dependability and Performance Evaluation of Intrusion-Tolerant Server Architectures *

Vishu Gupta, Vinh Lam, HariGovind V. Ramasamy,
William H. Sanders, and Sankalp Singh**

Coordinated Science Laboratory,
University of Illinois at Urbana-Champaign
1308 W. Main Street, Urbana, IL 61801, USA
{vishu, lam, ramasamy, whs, sankalps}@crhc.uiuc.edu

Abstract. In this work, we present a first effort at quantitatively comparing the strengths and limitations of various intrusion-tolerant server architectures. We study four representative architectures, and use stochastic models to quantify the costs and benefits of each from both the performance and dependability perspectives. We present results characterizing throughput and availability, the effectiveness of architectural defense mechanisms, and the impact of the performance versus dependability tradeoff. We believe that the results of this evaluation will help system architects to make informed choices for building more secure and survivable server systems.

1 Introduction

Intrusion tolerance [6] is an approach to handling malicious attacks, in which the impracticability of making a system fully secure against all attacks is recognized and intrusions are expected, but the system is designed to provide proper service in spite of them (possibly in a degraded mode). Intrusion tolerance has the potential to become a very useful approach in building server architectures that withstand attacks. Several such intrusion-tolerant server architectures have been conceived in both academia and industry, including KARMA [7], ITSI [14], ITUA [3], and PBFT [2]. However, there has not been any comparative study of their performance and dependability. There are many challenges in doing such a study. First, it is difficult to identify representative architectures that cover the various design possibilities for building intrusion-tolerant architectures. Second, the problem of coming up with detailed yet reasonably high-level models of chosen representative architectures that could be comprehensively evaluated is a fairly complex one. The models should represent the design differences between architectures without getting tied down to low-level details. Third, coming up with appropriate measures that bring out the relative strengths and weaknesses of the representative architectures is a complex problem in itself.

In this paper, the above challenges are addressed for the first time (to the best of our knowledge), and a fairly comprehensive comparison of intrusion-tolerant server architectures is presented. We realize that given the many variations in implementing intrusion-tolerant systems, any comparative study is feasible only

* This research has been supported by DARPA contract F30602-00-C-0172.
** Names are in alphabetical order. Authors made equal contributions to the research.

if we identify *classes* of intrusion-tolerant architectures and limit our comparison to abstract architectures that are representative of these classes. In this work, we identify four classes of intrusion-tolerant server architectures based on how requests are handled and how decisions are made in response to intrusions. In modeling the effectiveness of these classes of intrusion-tolerant architectures, we realize that the performance and dependability of these intrusion-tolerant systems cannot be quantified in a deterministic manner, because the systems do not provide complete immunity to all possible intrusion methods. An attractive option for evaluating intrusion-tolerant systems is via probabilistic modeling [11], as shown by Singh et al. [15], who validated an intrusion-tolerant replication system, with variations in internal algorithms, using probabilistic models.

In this paper, we evaluate and compare the strengths and weaknesses of the four architectures in probabilistic terms. We use Stochastic Activity Networks (SANs) [12] as our representation of the models for the architectures. By varying the parameters of the models, we obtain information about performance and intrusion tolerance characteristics of the different architectures.

2 Intrusion-Tolerant Server Architectures

We consider intrusion-tolerant architectures that follow a *client-service system* paradigm (for example, a web browser as a client and a collection of web servers as the service system). All such systems are based on replication of information across a set of servers, and rely on a distributed architecture that routes incoming requests among several server nodes in a user-transparent way. All such systems also have some mechanism by which the incoming requests are spread among the servers. We consider only those mechanisms for routing the requests among the server nodes that do not require the clients to know that there are replicated servers in the service system and that do not divulge any information about which of the replicated servers actually service a particular client's request. This "hiding" of the servers from clients is necessary for anonymity and security purposes. Client-based, DNS-based, and server-based routing mechanisms (see [1] for a detailed classification of the various approaches for routing requests among multiple servers) do not satisfy the requirement of "hiding." The appropriate routing mechanism is the dispatcher-based approach, in which a single virtual IP address is used for the entire service system. The dispatching mechanism could be centralized, in which case it would route requests to individual servers, or it could be logically distributed among the servers, in which case the requests would be multicast to the servers.

We explored the design space for intrusion-tolerant systems that satisfy the above criteria, and identified the following dimensions along which architectures can vary: (1) how the client requests get routed to the servers, (2) whether the decisions to reconfigure the system in response to intrusions are made centrally or in a distributed manner, and (3) whether multiple requests are served concurrently by different servers. Based on the above, we partitioned the design space into four classes. In this paper, we model four abstract architectures, each

of which is representative of one of those classes. All four architectures that we evaluate have the following components in one form or another:

Client: The client is a program, like a web browser, that establishes connections to the service system in order to satisfy user requests.

Service: This component implements the protocols to service an incoming client request. For example, it could be an HTTP server.

Intrusion Detector: This component could be a combination of multiple third-party intrusion detection tools and protocol-specific intrusion detection (in which violations of the protocol specification are treated as intrusions).

Configuration Manager Daemon: The Configuration Manager Daemon (or CMDaemon for short) uses the Intrusion Detector component to keep track of whether or not the service has been compromised, and implements strategies for recovering from attacks. There is one CMDaemon component for each Service component. Each CMDaemon monitors one Service component and may run in the same host as that Service component.

Configuration Manager: The Configuration Manager receives reports from the CMDaemons about the well-being of the Service Components that they monitor. It decides how to recover when an intrusion is reported, and instructs the CMDaemons about this decision. Each CMDaemon then implements those instructions in their respective Service components.

Gateway: This is the component whose IP address is known to the clients as the IP address of the service system. It serves as the dispatcher that controls the routing of the client requests to the Service components, helping to mask the identities of the Service components' operating systems and the service application. In architectures that do not have the Gateway component, all the servers receive all the client requests. That is done in various ways; for example, all the servers could be configured to be members of an IP multicast group. Clients would send their requests to this multicast address.

Firewall: This component filters incoming requests based on certain policies.

Database: The Database component is the store for the information that clients want to access. In this paper, we are not concerned about the exact organization of this component. Interested readers are referred to [5].

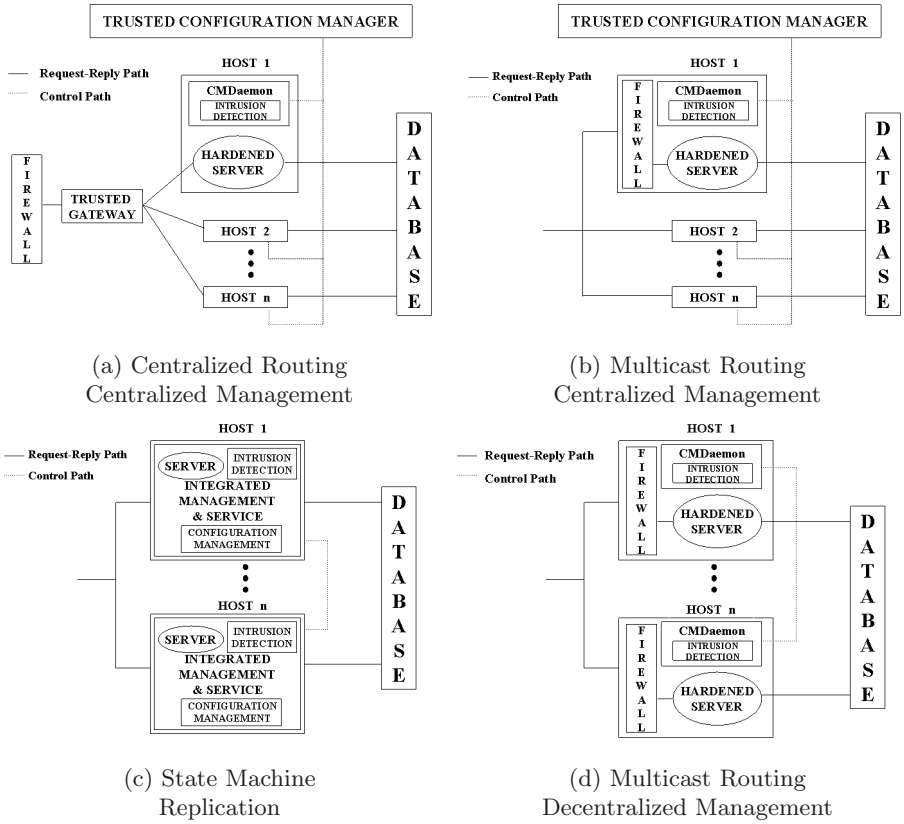
The four architectures differ in how the above components interact with each other, their placement, and which of them are trusted. A "trusted" component is one that is assumed not to fail. We now describe each of the four architectures in more detail.

Centralized Routing Centralized Management (CRCM) The goal of the CRCM design is to employ a small number of trusted components to protect a large set of servers and databases. In this design, a Firewall component filters the incoming requests, looking for signatures of commonly known attacks. The Gateway is a trusted component. An incoming request passes through the Firewall to reach the Gateway, which then forwards the request to a randomly chosen server from the active server set. The Gateway also masks server-specific and OS-specific information from all the replies. The service system consists of a large collection of servers. They share the same filesystem, but may run differ-

ent operating systems and different web-server software versions. In addition to the server software, each host that is part of the service system also runs a CMDaemon, which is responsible for detecting attacks via various mechanisms (e.g., integrity-checking of various critical files and checking of the process states). The CMDaemons report the health of the local server to the Configuration Manager, which is a trusted component. The Manager continually checks the integrity of the CMDaemons. If there is an intrusion detection, the Manager cleans the server state, and could roll back the potentially erroneous transactions committed by the intruded server. The Manager informs the Gateway about the current active server set. The Gateway uses that information in the selection of servers to process client requests.

Multicast Routing Centralized Management (MRCM) The MRCM architecture achieves intrusion tolerance through hardened, heterogeneous platforms. This hardening is achieved by embedding firewalls in each server host, and having extensive alert and intrusion-detection capabilities in each server host. Those capabilities form the CMDaemon component. There are no additional front-end firewalls like those in CRCM. Scalability is achieved through the ability to add additional platforms easily, and maintainability is achieved through the ability to remove and service platforms easily. All the servers receive all the requests sent to the single virtual IP address of the service. The service rules on each server determine what traffic to process and what to throw away. For example, rules could be based on the source IP address of the client. In essence, those service rules form a load-balancing policy. The load-balancing policy could be changed at the behest of the Configuration Manager (for example, when an intrusion is detected and the intruded host shut down), and the clients previously serviced by the intruded host would need to be distributed among the correct hosts. When an intrusion is detected, the Configuration Manager could instruct the servers to implement the new load-balancing policy by giving them an updated set of service rules. Through the CMDaemon on a host, the Configuration Manager could also update the filtering policies on the host-embedded firewalls so that traffic from specified clients is blocked or audited.

State Machine Replication (SMR) The SMR architecture employs a state-machine-replication-based approach [13] that tolerates malicious faults. A replication protocol that tolerates Byzantine faults, similar to [2], could be used (with some modifications to ensure user transparency) for this architecture. The requirement for an algorithm tolerating Byzantine faults is that it must have at least $3f + 1$ servers, where f is the number of simultaneous faults that need to be tolerated. SMR does not require an extensive firewall like those in the CRCM and MRCM architectures. Unlike CRCM and MRCM, there is no centralized trusted Configuration Manager and local CMDaemons. Instead, the Configuration Management is now distributed among the servers. The distributed Configuration Management and Service components are integrated into one logical unit. This integrated Management and Service unit is replicated across the set of servers, and the Byzantine-fault-tolerant protocol ensures that all correct servers maintain consistent state information for this integrated unit. As in MRCM, all



(a) Centralized Routing Centralized Management

(b) Multicast Routing Centralized Management

(c) State Machine Replication

(d) Multicast Routing Decentralized Management

Fig. 1. Architecture Block Diagrams

Table 1. Summary of the design features of the four architectures

Feature	CRCM	MRCM	SMR	MRDM
Parallelism in processing requests	Yes	Yes	No	Yes
Strict correctness of replies guaranteed	No	No	Yes	No
Configuration Manager	Centralized	Centralized	Distributed	Distributed
Required number of servers for uninterrupted service when f servers are compromised	f+1	f+1	$3f+1$	$3f+1$
Forwarding of client request by Gateway	to a randomly selected server	to all servers	to all servers	to a randomly selected server
Servicing of request	by the randomly selected server	based on source IP	by all servers	by the randomly selected server
Trusted components	2	1	0	0

requests reach all the servers. The set of servers processes one request at a time. The servers agree on the reply to be sent to the client, as well as on any updates to be made to the back-end database, through a Byzantine agreement protocol. SMR ensures that all replies sent to clients and updates made to the database are correct, as long as there are no more than f simultaneous corruptions in the system (we call this the *Byzantine agreement requirement*), but involves a large performance overhead due to the fact that all the requests are serialized and processed by the entire set of servers one at a time.

Multicast Routing Decentralized Management (MRDM) The MRDM design is a hybrid of the previous 3 architectures, and tries to achieve a trade-off between the better throughput performance achieved by the parallelism of the CRCM and MRCM architectures, and the strict correctness achieved by the SMR architecture, without relying on any trusted components. It does so by separating the service component in the SMR architecture from the configuration management. As in the SMR architecture, the Configuration Manager is distributed across the host nodes. However, unlike in SMR, the server nodes do not all process the same request at the same time. A firewall component embedded in each host (similar to the one in MRCM) could be used to filter out incoming requests based on specified policies. The incoming request is randomly routed to one of the servers (like in CRCM). Each host runs a server component and a configuration management component (which represents an integrated Configuration Manager, CMDaemon, and Intrusion Detector component). The servers can process requests independently from each other (unlike in SMR), but the configuration management components across all the hosts coordinate with each other, distribute knowledge about intrusions, and come to agreement about the configuration changes that need to be made in response to intrusions. At the core of the configuration management component could be an intrusion-tolerant group membership protocol (such as the one in [10]) that requires the participation of at least $3f + 1$ nodes to tolerate f simultaneous faults. By separating the service component from the management component, we are able to retain the parallelism of the CRCM and MRCM architectures, and by distributing the management component, we remove the need for having a central trusted Configuration Manager. However, MRDM does not guarantee strict correctness of replies (as SMR does), since the intruded node could still be servicing some requests, and potentially sending erroneous replies, during the time period between the intrusion of a node and the detection of the intrusion. The SMR architecture, on the other hand, masks the effects of a subset of intruded servers, as long as the threshold requirement of f is satisfied.

2.1 Assumptions and Attack Model

We assume *staged* attacks, which means that there is a non-negligible time between successive node infiltrations. That gives the defense some time to react. None of the above architectures can defend against a situation in which all the hosts are simultaneously intruded. They also cannot defend against a situation

in which the attacker intrudes the various nodes in stages, but the compromised nodes show no observable signs of an intrusion until all the nodes have been intruded (this is essentially the same as the first situation). For the staged attack assumption to be true, node failures must not be strongly correlated. That could be achieved, for instance, by running different implementations of the service code and/or the operating system.

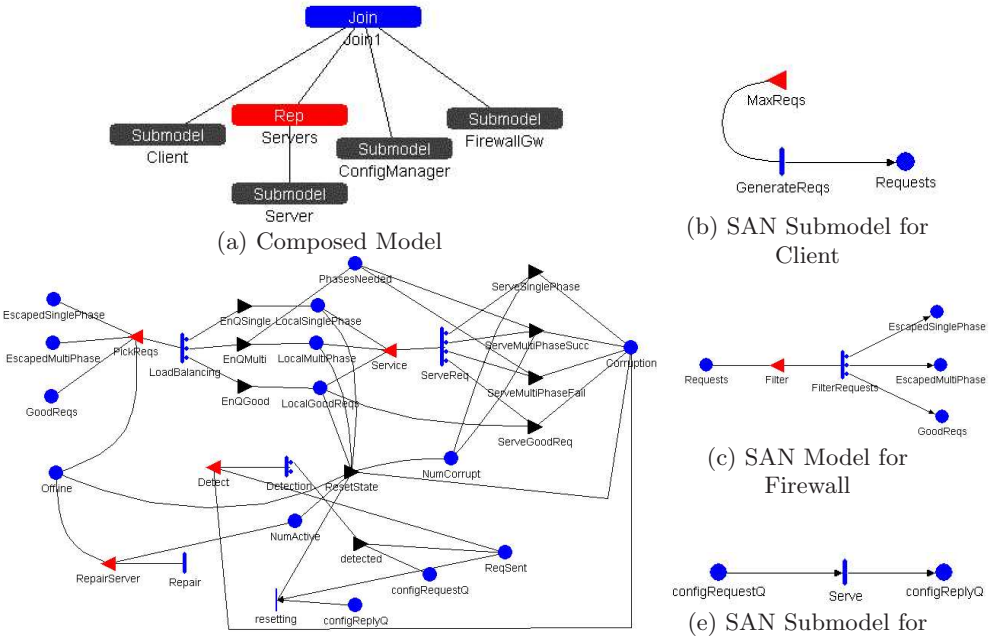
Within the staged attack model, there could be two kinds of attacks on a single host: *multi-phase attacks* that require a sequence of attacks in order to successfully compromise the host (for example, an attacker could upload a file line-by-line using the Windows "echo" command), and *single-phase attacks* that successfully compromise the host in one shot (for example, the attacker could guess the correct password and gain root access on the first attempt).

The CRCM and MRDM architectures employ *dispersion*, i.e., because of the random selection of servers by the Gateway, requests from the same client could be processed by different servers. That decreases the probability that different phases of a multi-phase attack will reach the same server. That, in turn, increases the time required to exploit any single web server using multi-phase attacks.

3 SAN Models for the Intrusion-Tolerant Architectures

Stochastic Activity Networks, or SANs, are a convenient, graphical, high-level language for capturing the stochastic (or random) behavior of a system. A SAN has the following components: *places* (denoted by circles), which contain tokens (the term "marking" is used to indicate the number of tokens in a place) and are like variables; *tokens*, which indicate the "value" or "state" of a place; *activities* (denoted by vertical ovals), which change the number of tokens in places; *input arcs*, which connect places to transitions; *output arcs*, which connect transitions to places; *input gates* (denoted by triangles pointing left), which are used to define complex enabling predicates and completion functions; *output gates* (denoted by triangles pointing to the right), which are used to define complex completion functions; *cases* (denoted by small circles on activities), which are used to specify probabilistic choices; and *instantaneous activities* (denoted by vertical lines), which are used to specify zero-timed events. An activity is enabled if for every connected input gate, the enabling predicate contained in it is true, and for each input arc, there is at least one token in the connected place. Each case has a probability associated with it and represents a probabilistic choice of the action to take when an activity completes. When an activity completes, one token is added to each place connected by an output arc, and functions contained in connected output gates and input gates are executed. The output gate and input gate functions are usually expressed using pseudo-C code. The times between enabling and firing of activities can be distributed according to a variety of probability distributions, and the parameters of the distribution can be a function of the state.

We have modeled the four architectures described in Section 2 as composed SANs. Atomic models were built for various components of each architecture,



(a) Composed Model

(b) SAN Submodel for Client

(c) SAN Model for Firewall

(d) SAN Submodel for Server

(e) SAN Submodel for ConfigManager

Fig. 2. SAN Models for CRCM

and complete models were then built using replicate and join operations. The salient features that we have modeled for each architecture include generation of client requests and attacks, organization of firewalls and filtering of requests, organization of servers and distribution of requests to servers, servicing of requests and effect of attacks, detection mechanisms, system reconfiguration upon detection of corruption, and repair of affected components. We have used exponential distribution for the timed activities in all the models. We believe this is a realistic assumption, because the request arrival process and servicing of requests by servers (especially web servers) are largely memoryless, and hence are well-represented by exponential inter-arrival times and exponential service times. Single-phase attacks and the subsequent phases in a given multi-phase attack are generated with some probability on the incoming requests; hence, they also have an exponential distribution in our SAN models. We developed that approach in order to keep the attack model fairly simple; we focused the complexity in the models to reveal the differences among various architectures. We understand that we may need sophisticated attack models in order to model the intrusion response behavior of the architectures more accurately. That may be the focus of another study. Due to space limitations, here we provide only a high-level description of the models of the individual architectures. A much more elaborate description of the SAN models is presented in [8].

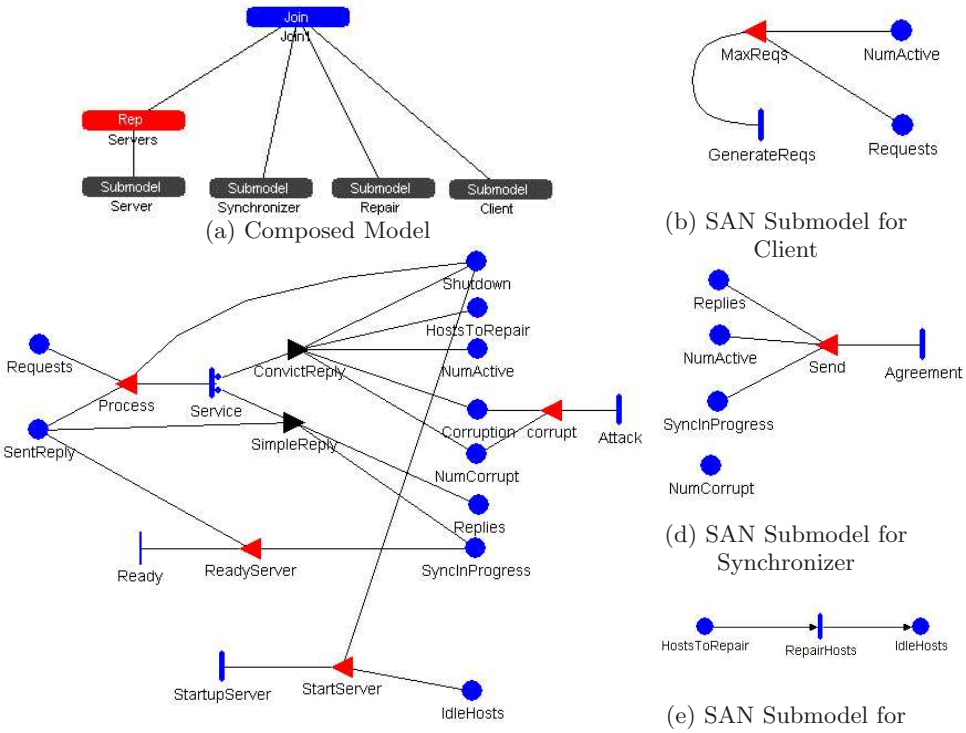
Centralized Routing Centralized Management (CRCM) The composed model for CRCM (Figure 2(a)) consists of four atomic SAN submodels: **Client**, **Server**, **ConfigManager**, and **FirewallGw**. The **Server** submodel

is replicated `NumServers` times, where `NumServers` is a global variable indicating the number of hosts running servers. Since requests have to pass through a firewall and a gateway before they are distributed to individual servers, we have a single unreplicated **Client** SAN (Figure 2(b)) to model the generation of incoming requests from the clients.

The **FirewallGw** SAN in Figure 2(c) models the firewall that filters incoming requests with known attack signatures. We model general attacks, including the ones that are not malformed client requests, as a part of the request stream. That is acceptable, since the request stream models the path all attacks follow (all packets pass through the firewall to reach the servers), and since effects of single-phase and multi-phase attacks are similar (they result in corruption of a server).

The **Server** SAN in Figure 2(d) models the centralized distribution of client requests to individual servers, servicing of requests, corruption of servers due to attacks, dispersion of multi-phase attacks, and detection of corruption and the system's response to it. The local place *Corruption* keeps track of the level of corruption of this server. A marking of 0 implies no corruption at all, and a marking of `MaxPhases` implies complete corruption, which is sufficient to influence the server's behavior. A value in between indicates that some phases of a multi-phase attack have been successful, but that the system is not corrupt enough to behave incorrectly. We model dispersion by having the probability of success of a phase in a multi-phase attack be the reciprocal of the marking of *NumActive*, a shared place that keeps track of the number of servers online. That accurately models the fact that each phase randomly goes to any of the active servers. The probability of successful detection is proportional to the number of changes that have been made to the configuration of the server (represented by the number of successful attack phases, which is equal to the marking of *Corruption*). Because of model size and complexity, we do not model false alarms. However, that does not constitute a shortcoming of our models, given that our focus in the models is on the *effect* of intrusion reports. Hence, we model a composite of actual attacks and false alarms (or, equivalently, correct and false intrusion reports). Upon successful detection, the Configuration Manager causes the server to be taken offline. The Manager informs the load-balancing gateway about this change, and the latter no longer forwards new requests to the server. The activity *Repair* represents the process of reinitializing the state of the server, after which the server can receive requests again.

Multicast Routing Centralized Management (MRCM) The composed model and atomic SAN submodels for MRCM are similar to those for CRCM. Here we point out the major differences. Since a firewall is now present on each host running the server, the **FirewallGw** and **Server** submodels are joined to form a model of each host. The resulting submodel is replicated `NumServers` times to form a model of the set of servers. Requests for each server are generated separately; there is no centralized request generation as in CRCM. This is done to model each request going to all servers, and exactly one of them picking it up



(a) Composed Model

(b) SAN Submodel for Client

(c) SAN Submodel for Server

(d) SAN Submodel for Synchronizer

(e) SAN Submodel for Repair

Fig. 3. SAN Models for SMR

for service, while others discard it. We model the redistribution of requests when a server goes offline by setting the rate of *FilterRequests* to be weighted by the fraction of the total number of servers that are currently active. Since there is no dispersion in this architecture, if the case corresponding to multi-phase attack is chosen in *ServeReq*, the phase is always successful, resulting in an increase in the marking of *Corruption*.

State Machine Replication (SMR) The composed model for SMR (Figure 3(a)) consists of four atomic SAN submodels: **Client**, **Server**, **Synchronizer**, and **Repair**. The **Client** SAN in Figure 3(b) models the centralized generation of incoming requests to the system, since each request is sent to all the active servers. The **Server** SAN (Figure 3(c)) models the processing of client requests by a server, attacks on a server, performance of Byzantine agreement between servers before a reply is sent back to the client, exhibition of incorrect behavior by corrupt servers, the subsequent exclusion of corrupt servers from the server group (provided there are enough uncorrupted servers for agreement), restarting of new servers on standby hosts, and repair of excluded hosts. Since the system reacts identically to single-phase and multi-phase attacks (since each request is sent to all servers), we have modeled both by a single activity. Also, since each server has a publicly visible IP address and there is no fire-

wall, we have modeled the attack generation explicitly, instead of having it be a part of the request stream. On firing, the marking of the local place *Corruption* is set to 1, and the marking of the shared place *NumCorrupt* is incremented. The activity *Service* represents the processing of a client request by the server, and the reaching of Byzantine agreement among the servers on the reply. If the marking of *Corruption* is 1, the probability of the case corresponding to the output gate *ConvictReply* is `probMisbehavior`, a global variable that represents the probability that a corrupt replica will exhibit corrupt behavior during the agreement process. Upon misbehavior, the server is taken offline, and the marking of the shared place *HostsToRepair* is incremented, since the host on which the server was running is also excluded, and we need to repair this host and bring it back into the system. If the marking of *Corruption* is 0, the case corresponding to the output gate *SimpleReply* is chosen with a probability of 1. The activity *StartupServer* represents the starting of a new server on a standby host, to replace one that has been shut down. We include standby hosts for SMR, because Byzantine agreement among hosts is the only way of detecting corruption, and it is necessary to have the corrupt server replaced quickly (by a server running on a standby host) to maintain the same level of intrusion tolerance. The SAN representation of the **Synchronizer** submodel (Figure 3(d)) models the completion of the response to a client request. It is needed since the servers have to maintain the same state. The SAN in Figure 3(e) models the repair process of the excluded hosts, which results in their transition to the standby state.

Multicast Routing Decentralized Management (MRDM) The composed model and atomic SAN submodels for MRDM are similar to those for MRCM. The major differences are as follows. The composed model for MRDM does not have a **ConfigManager** submodel, since the management decision is taken in a decentralized manner using Byzantine agreement; in the **Server** submodel for MRDM, upon detection, a corrupt server is taken offline only if the other servers can reach a Byzantine agreement on shutting it down. Since multiphase attacks are dispersed in MRDM, the probability of success of an attack phase in the **Server** submodel varies inversely with the number of active servers.

4 Results

We used the Möbius [4] tool to build the SANs, define performance and intrusion tolerance measures, design studies on the models, simulate the models, and obtain values for the measures defined on various studies. The measures defined on each model for use in the studies are as follows:

Productive Throughput: This measure characterizes the number of requests that the system replies to correctly per time unit. We assume that all correct servers reply correctly to the requests they receive, and all corrupt servers reply incorrectly to the requests they receive. We study the expected value of this measure averaged over a time interval.

Unproductive Throughput: This measure characterizes the number of requests that the system replied to incorrectly per time unit.

Strong Unavailability for an interval: This measure characterizes the fraction of time the service was *improper* in the given time interval. For this measure, the service was defined to be improper (for the CRCM, MRCM, and MRDM architectures) if at least one active server was in a corrupt, undetected state, or all servers were offline for repair. For SMR, the service is improper if more than a third of the active servers are corrupt. Hence, a strongly available system does not send an incorrect reply to any request.

Weak Unavailability for an interval: Here, we use a weaker definition of proper service. The service is proper if at least one correct server is online. This measure is not defined on models for SMR. The above two unavailability measures characterize the survivability of the systems as perceived by a user.

Fraction of Corrupt Servers: This measure characterizes the fraction of active servers that are corrupt at a given instant of time.

We designed several studies on the models to determine how various architectures behave when we vary some important system parameters, and to determine the range of parameter values for which a particular architecture is superior over others, with respect to intrusion tolerance and performance characteristics. The input parameters we varied are the number of hosts in the system, the rate of single-phase attacks on the system, the rate of multi-phase attacks on the system, the quality of the detection mechanism being used, and the rate at which components taken offline are repaired and brought back into the system.

Unless otherwise specified, we used the values given below for various input parameters. We need to emphasize here that the reader need not be particularly concerned about our specific choice of parameter values, because the aim of these experiments is to present performance and dependability *trends/patterns* of these architectures relative to each other, rather than exact values. It is very hard (if not impossible) to come up with any single universally applicable choice of values, because these architectures could be deployed in widely varying situations. However, using our SAN models, we can quite easily conduct these experiments for a large range of parameter values.

We consider a time unit of one minute. Request arrival rate was set to 100 requests (to the entire service system) per minute for all the architectures. Cumulative attack rates were set to be 12 and 6 per hour for single and multi-phase attacks respectively.

The local detection components running on each server check for corruption once every two minutes for CRCM, and once every minute for MRCM and MRDM. That is justified because CRCM uses a centralized detection mechanism with lightweight daemons running on individual hosts, resulting in slower detection, whereas all the detection in MRCM and MRDM is done locally on each host, resulting in faster detection. The probability of detecting a corruption in each run is set to 0.5. Likewise, in SMR, a corrupt server misbehaves with a probability of 0.5. (In Section 4.1, we explain why the probability of misbehavior in SMR is equivalent to the probability of detection in other architectures.)

The probability that the centralized firewall in CRCM will detect and filter out an attack in CRCM was set to 0.75. The probability that the local firewalls on each host running a service component in MRCM and MRDM will detect and filter out an attack was set to 0.4. We use a higher probability for CRCM since it has a centralized firewall running on a dedicated machine that can detect and filter out attacks more intelligently. However, we realize that the exact degree of difference in a real setting will vary depending on the strength of firewalls actually deployed.

The mean time to repair an offline server was set to 17 minutes in all the architectures.

The total number of hosts was set to 12. So that all architectures would have similar amounts of resources, that number includes the hosts running service components as well as the hosts running trusted components. Hence, CRCM had 10 hosts running service components and 2 hosts running trusted components (the Configuration Manager and Gateway); MRCM had 11 hosts running service components and one host running a trusted component (the Configuration Manager); and SMR and MRDM each had all 12 hosts running service components. SMR had 3 additional hosts in the standby state.

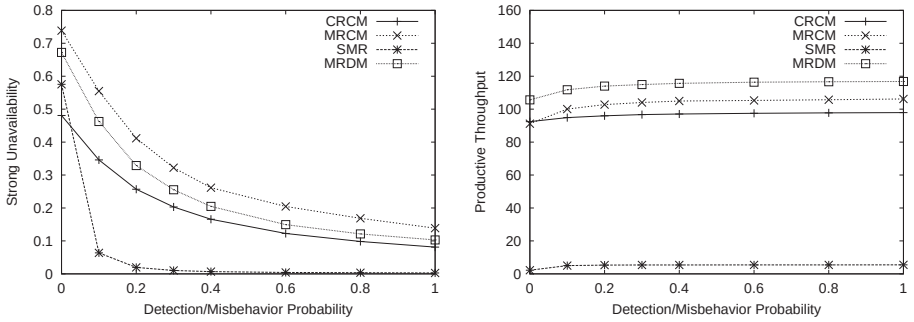
The time interval considered is [0, 30 minutes]. The fraction of corrupt servers is measured at the end of this interval.

We used simulation to solve all the models; all results presented here have a 95% confidence interval.

4.1 Comparison under Varying Quality of Detection

For the CRCM, MRCM, and MRDM architectures, the *quality of detection* is the probability with which an intrusion detection system can ascertain that a system has been compromised, given that the system is actually corrupt. SMR does not have a separate intrusion detection system, and detects intrusion primarily through Byzantine agreement by the group; the group members can know a corrupted member is corrupted only when it shows some misbehavior during the agreement, by deviating from the protocol specification. That is modeled by the probability of misbehavior. We varied the detection probability from 0.0 (no intrusion detection) to 1.0 (perfect intrusion detection). For SMR, the probability of misbehavior was varied from 0.0 (corrupt server does not misbehave at all) to 1.0 (corrupt server always misbehaves).

Figure 4(a) shows that in the absence of an intrusion detection mechanism (or equivalently, absence of misbehavior in SMR), the strong unavailability of any architecture depends primarily on the architecture's defense against intrusion attempts. Thus, CRCM shows the best performance and the least unavailability, because it has a strong firewall and better handling of multi-phase attacks. All the other architectures suffer because of weaker firewalls; MRCM performs the worst because it is most susceptible to multi-phase attacks, due to lack of dispersion. When the probability of detection increases, all architectures become more available, but among CRCM, MRCM, and MRDM, the CRCM architecture remains the best and MRCM the worst for the same reasons. We notice that



(a) Strong Unavailability (b) Productive Throughput

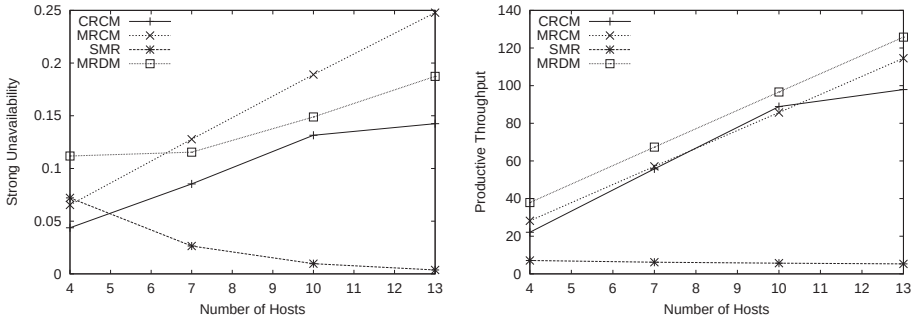
Fig. 4. Varying Detection/Misbehavior Probability

SMR is initially very sensitive to any increase in the probability of misbehavior, because as long as the Byzantine agreement requirement is met, any corrupt misbehaving servers can be immediately eliminated. However, for large values of the misbehavior probability, it becomes increasingly difficult for more than one-third of the group to be corrupt at any one time (which is the criterion for unavailability in SMR).

Figure 4(b) shows that SMR has the least amount of productive throughput, because all servers process every request. Its throughput does not change for misbehavior probabilities greater than 0.3, because above that value it is almost always available. A trend that is observed in all architectures is that beyond a certain detection probability (approximately 0.3 for the input parameter values used in this study), throughput does not show an appreciable increase. The reason is that throughput depends primarily on the system's total service capacity (given by the service rate) and the arrival rate, and these parameters were kept constant in our studies. Among the CRCM, MRCM, and MRDM architectures, the differences in productive throughput is due to the fact that MRDM has two more servers than CRCM and one more server than MRCM.

4.2 Comparison under Varying Numbers of Hosts in the System

Varying the number of hosts in the system from 4 to 13 implies that the number of hosts serving requests (servers) varies from 2 to 11 in CRCM, from 3 to 12 in MRCM, and from 4 to 13 in SMR and MRDM. For 4 hosts, SMR and MRDM are more unavailable than CRCM and MRCM (see Figure 5(a)), because they require Byzantine agreement in order to exclude corrupt servers, and 4 servers can tolerate at most one corruption. Given enough time, it may be easy to corrupt one server, and beyond that point, no further corruptions can be tolerated, hence affecting availability. Also, MRDM performs worse than SMR, because MRDM is considered unavailable in the strong sense even when one server is corrupt, while SMR is considered available until one-third of the servers are corrupt. SMR shows decreasing unavailability with an increasing number of hosts, because larger group size enables it to tolerate a larger number



(a) Strong Unavailability (b) Productive Throughput

Fig. 5. Varying Number of Hosts

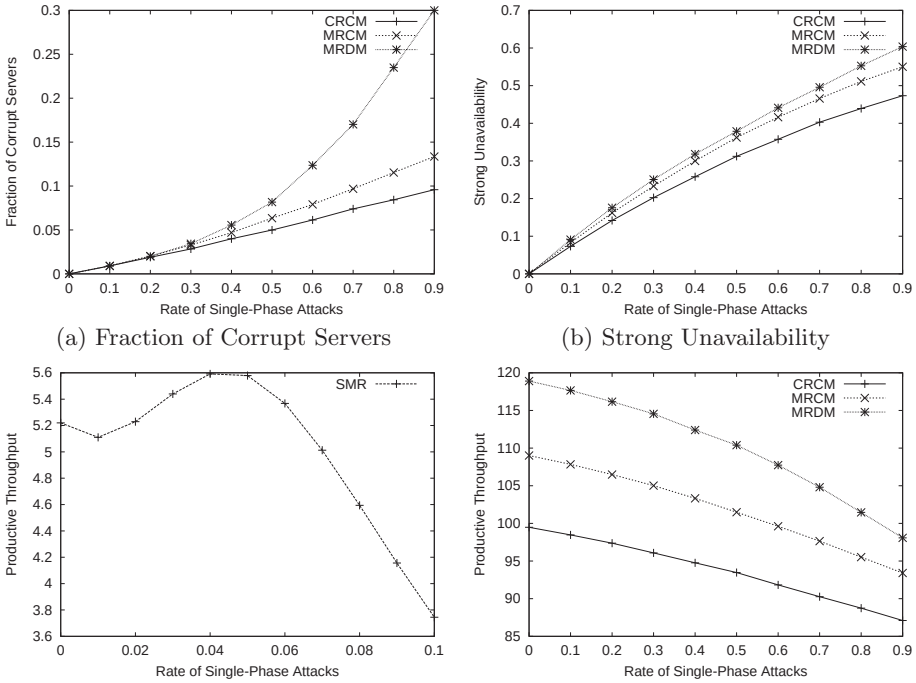
of simultaneous faults. However, unavailability for CRCM and MRCM increases with the number of hosts; that may seem counter intuitive, but the greater number of hosts means that there is a greater chance that one host will be corrupt and online. Like SMR, a larger number of servers makes it easier for MRDM to detect corrupt servers and exclude them. On the other hand, a larger number of servers makes it more likely that MRDM will have a corrupt server online. Because of these opposing forces, MRDM's unavailability initially remains unchanged, and starts increasing later, because the negative effect of having more servers becomes more dominant. We also note that CRCM does not show an appreciable increase in unavailability above 10 hosts. The reason is that for the chosen arrival rate and service rate of the individual servers, the waiting time for any request (and hence any attack) is negligible for 10 hosts, and is unaffected by a further increase in the number of hosts.

In SMR, all hosts process every request, so increasing the number of hosts does not help in increasing throughput; rather, productive throughput (Figure 5(b)) actually falls a little, because of an increase in agreement delays. MRCM and MRDM show steady increase in productive throughput, which is to be expected from parallel processing architectures. On the other hand, CRCM does not show an appreciable increase in productive throughput when the number of hosts goes beyond 10, because at that point the central dispatcher starts acting as a bottleneck in the system, as mentioned before.

4.3 Comparison under Varying Single-phase Attack Rates

In this study, we varied the probability that an incoming request is a single-phase attack from 0 to 0.009 (in increments of 0.001) for the CRCM, MRCM, and MRDM architectures. That resulted in the single-phase attack rate varying from 0 to 0.9 (in increments of 0.1), since the request arrival rate is 100. The probability of multi-phase attacks was set to 0. For SMR, the attack rate was varied along the same lines.

Figure 6(a) shows the variation in the fraction of active servers that are corrupt for the CRCM, MRCM, and MRDM architectures. We observe that



(a) Fraction of Corrupt Servers

(b) Strong Unavailability

(c) Productive Throughput for SMR

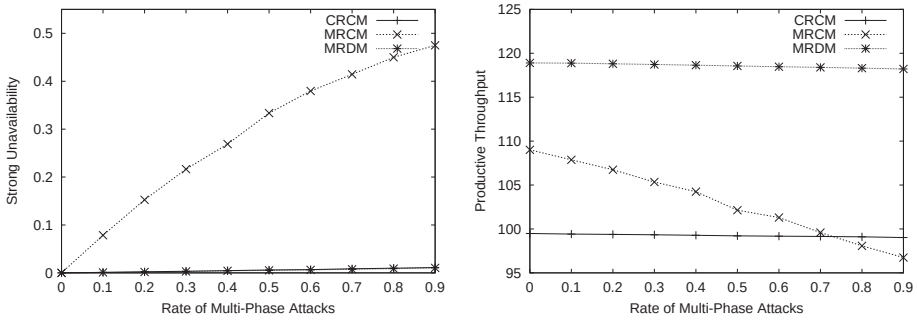
(d) Productive Throughput for CRCM, MRCM, and MRDM

Fig. 6. Variation in Measures with Varying Single-phase Attack Probability

CRCM performs better than the other two architectures. That can be attributed to CRCM's stronger centralized firewall as compared to the weaker local firewalls in MRCM and MRDM. Since dispersion of multi-phase attacks is not a factor in this study, MRCM performs comparably. The linear increase for CRCM and MRCM is as expected, but there is a rapid deterioration for MRDM. The reason is that in MRDM, for higher attack rates, there is a significant probability that more than a third of the servers will become corrupt before any detection, thus violating the Byzantine agreement requirement, and hence making it impossible for any corrupt server to be removed from the set of active servers.

Figure 6(b) shows the variation in strong unavailability for the CRCM, MRCM, and MRDM architectures. All the architectures perform similarly and are strongly affected by the rate of attacks. CRCM is slightly better due to its strong centralized firewall, and MRDM is slightly worse due to the failure of the Byzantine agreement algorithm for higher attack rates.

Figure 6(c) depicts the variation in productive throughput for the SMR architecture. The performance overhead due to the Byzantine agreement protocol increases with the number of servers in the system. However, instead of increasing linearly, it increases as a step function, with almost fixed-size jumps whenever the number of servers is of the form $3f + 1$ (i.e., jumps at 4, 7, 10, and so on). This has been shown experimentally in [10]. Since the throughput varies



(a) Strong Unavailability (b) Productive Throughput

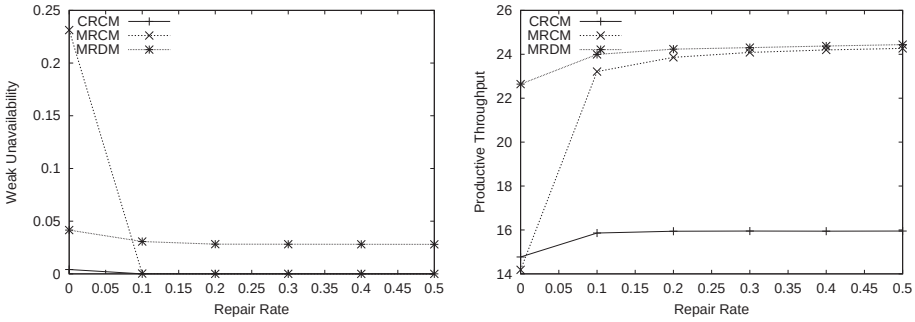
Fig. 7. Variation in Measures with Varying Multi-phase Attack Probability

inversely with the delay, the gain in throughput with decrease in the number of servers is more substantial when the number of servers is smaller. Increasing the attack rate decreases the number of servers; this decreases the Byzantine agreement overhead, and hence tends to increase the throughput. On the other hand, the probability of enough servers becoming corrupted to violate the Byzantine agreement requirement increases with increasing attack rates, hence decreasing productive throughput. The nature of this graph can be attributed to the competition between these two opposing forces. The former dominates the initial portion of the graph, while the latter dominates when the attack rate is higher. As explained above, the gain in throughput is not much when the expected number of servers online is high, and that leads to the domination of the latter force for very low attack rates, resulting in the initial dip in the graph.

Figure 6(d) shows that productive throughput decreases with increasing attack rates, as fewer correct servers are online. The relative performance of the architectures can be explained by the facts that CRCM has a performance bottleneck of centralized request routing, and that MRDM, MRCM, and CRCM have 12, 11, and 10 servers working in parallel, respectively.

4.4 Comparison under Varying Multi-phase Attack Rates

In this study, we vary the probability that a particular request is part of a multi-phase attack from 0 to 0.009, while keeping the number of single-phase attacks at 0. Figure 7(a) shows that CRCM and MRDM (coinciding lines) perform better than MRCM with respect to strong unavailability. The reason is that multi-phase attacks in CRCM and MRDM are largely unsuccessful due to dispersion, and have a negligible effect on strong unavailability. The effect on MRCM becomes more evident when we look at the productive throughput for the three architectures in Figure 7(b). Though MRCM starts out better than CRCM because of one additional server, its performance degrades rapidly as we increase the probability of multi-phase attacks.



(a) Weak Unavailability (b) Throughput up to $t = 120$ min

Fig. 8. Variation in Measures with Varying Repair Rates

4.5 Comparison under Varying Repair Rates

When a corrupted server is detected, it is removed from the set of active servers, taken offline, and put into repair. After repair, the server is put back into the pool of active servers. The system would eventually fail if there was no repair or the repair was not "fast enough," i.e., if the mean time between successful attacks is shorter than the average time taken to repair a server and put it back into service. Thus, we can intuitively predict that a faster repair rate is crucial for ensuring that the system provides continuous service.

Figure 8 confirms this intuition. In obtaining the data for these graphs, we considered, for all architectures, a set of 4 hosts running the service component. Additional hosts were used for the trusted management components (Gateway, Configuration Manager, and Firewall) if those components are required in the architecture. We varied the repair rate from 0 (no repair) to 0.5 (very fast repair rate: one repair every 2 minutes), while the other parameters were kept constant. The attack rate was kept constant at 0.08 per time unit. As the repair rate varies from 0 upwards, we can see that productive throughput increases until a saturation point. The saturation point is reached when the repair rate is faster than the attack rate. Increasing the repair rate beyond that point has some beneficial effects, but not substantial improvements. A similar trend can be observed from the graphs depicting weak unavailability. The saturation point (for a given estimate of the attack rate) represents the optimal repair rate; it is "optimal" in the sense of getting maximum benefit from minimal cost for repair.

From Figure 8(a), we can see that with no repair, CRCM performs the best, because of its strong firewall and its use of a dispersion mechanism. MRDM and MRCM do not have a strong firewall, but MRDM outperforms MRCM due to dispersion in the former. Since CRCM starts out with low unavailability, it is not affected substantially by an increase in repair rate. MRCM matches the low unavailability of the CRCM architecture after the optimal repair rate has been reached. The MRDM architecture, on the other hand, is not able to attain such low unavailability, even after the saturation point. The reason is that our experiments were conducted with 4 servers, and when the number of correct

servers drops to 3, it is not possible to reach Byzantine agreement to remove the next corrupted server from the set of active servers.

Though the CRCM and MRCM architectures outperform the MRDM architecture in availability, with respect to correctness of replies (productive throughput), MRDM is clearly superior (as seen from Figure 8(b)). The duration between detection of an intrusion and removal of the corrupted server from the active set is shorter for MRDM than for the CRCM and MRCM architectures, due to the fact that it does not have the bottleneck of a centralized manager. Therefore, the number of potentially erroneous replies that a corrupted server could send before being removed would be less for the MRDM architecture than for other architectures. However, we expect that for a greater number of servers, this advantage may become less important for MRDM, because the overhead due to the Byzantine agreement protocol increases significantly as the number of servers increases, as shown experimentally in [10].

5 Conclusion

This work is the first attempt to evaluate intrusion-tolerant server architectures. We define a series of relevant metrics and present a probabilistic evaluation and comparison of four representative intrusion-tolerant server architectures. The results present useful information about the intrusion tolerance and performance characteristics of the architectures, by means of varying system parameters such as the quality of intrusion detection, rate of attacks on the system, amount of resources, and time to repair an intruded server.

The results show that architectures that use a small number of trusted components to secure a large set of servers have better availability than architectures with no trusted components when the level of redundancy in the system is not very large. However, [9] shows that it is difficult, if not impossible, to implement truly trustworthy components. Such architectures also usually employ centralized decision-making, which is a potential performance bottleneck.

State-machine-replication-based architectures that employ Byzantine fault-tolerant protocols for agreement on the request processing have the best intrusion tolerance characteristics, but they have comparatively lower performance. Hence, such architectures are a good choice for implementing mission-critical systems for which the ability to withstand intrusions is more important than performance.

Architectures that employ decentralized decision-making and serve multiple requests in parallel have the best performance for a given amount of resources, since all the resources can be used for request processing. They are superior to centralized architectures, for which a portion of resources need to be set aside for hosting trusted components. However, from an intrusion tolerance perspective, the effectiveness of such decentralized architectures is realized only when there is a sufficient degree of redundancy. We also observe that introducing unpredictability in request routing (dispersion) is highly effective in defense against multi-phase attacks, and that it is critical that the mean time to repair be much less than the mean time between attacks.

We believe that our choice of values for model parameters is reasonable, but more importantly, our models allow system designers to evaluate alternative architectures by assigning different values for those parameters as they deem appropriate. This certainly enhances their ability to make more informed choices between various intrusion-tolerant architectures easily and quickly, before undergoing the expensive process of building and evaluating multiple prototypes.

Acknowledgments: We thank Dr. Marinho Barcellos for his help in improving the manuscript, and Jenny Applequist for her editorial assistance.

References

1. V. Cardellini, M. Colajanni, and P. S. Yu, "Dynamic Load Balancing on Web-server Systems," *IEEE Internet Computing*, Vol. 3, No. 3, pp. 28–39, 1999
2. M. Castro and B. Liskov, "Practical Byzantine Fault Tolerance," *Proc. Third Symp. on Operating Sys. Design and Implementation (OSDI '99)*, pp. 173–186, 1999
3. M. Cukier, J. Lyons, P. Pandey, H. V. Ramasamy, W. H. Sanders, P. Pal, F. Webber, R. Schantz, J. Loyall, R. Watro, M. Atighetchi, and J. Gossett, "Intrusion Tolerance Approaches in ITUA," FastAbstract in *Supplement of the 2001 International Conference on Dependable Systems and Networks*, pp. B64–B65, 2001
4. D. D. Deavours, G. Clark, T. Courtney, D. Daly, S. Derisavi, J. M. Doyle, W. H. Sanders, and P. G. Webster, "The Möbius Framework and Its Implementation," *IEEE Trans. on Software Engineering*, Vol. 28, No. 10, pp. 956–969, October 2002
5. A. Delis and N. Roussopoulos, "Performance and Scalability of Client-Server Database Architectures," *Proc. Intl Conf. in Very Large Data Bases (VLDB)*, pp. 610–623, 1992
6. Y. Deswarte, L. Blain, J. C. Fabre, "Intrusion Tolerance in Distributed Computing Systems," *Proc. IEEE Symposium on Security and Privacy*, pp. 110–121, 1991
7. Draper Laboratories, Inc., "Kinetic Application of Redundancy to Mitigate Attacks," *DARPA OASIS Program*, http://www.tolerantsystems.org/ProjectSummaries/IT_Using_Masking_Redundancy_and_Dispersion.html
8. V. Gupta, V. Lam, H. V. Ramasamy, W. H. Sanders, and S. Singh, "Dependability and Performance Evaluation of Intrusion-Tolerant Server Architectures," *CRHC Technical Report*, 2003, to appear
9. U. Lindqvist, T. Olovsson, and E. Jonsson, "An Analysis of a Secure System Based on Trusted Components," *Proc. Eleventh Annual Conf. on Computer Assurance (COMPASS '96)*, pp. 213–223, Gaithersburg, Maryland, 1996
10. H. V. Ramasamy, P. Pandey, J. Lyons, M. Cukier, and W. H. Sanders, "Quantifying the Cost of Providing Intrusion Tolerance in Group Communication Systems," *Proc. Intl Conf. on Dependable Sys. and Networks (DSN-2002)*, pp. 229–238, 2002
11. W. H. Sanders, M. Cukier, F. Webber, P. Pal, and R. Watro, "Probabilistic Validation of Intrusion Tolerance," FastAbstract in *Supplemental Volume of the 2002 International Conference on Dependable Systems and Networks*, pp. B78–B79, 2002
12. W. H. Sanders, and J. F. Meyer, "Stochastic Activity Networks: Formal Definitions and Concepts," In *Lectures on Formal Methods and Performance Analysis*, LNCS 2090, Springer-Verlag (E. Brinksma, H. Hermanns, J.P. Katoen, Ed.), Berlin, pp. 315–343, 2001
13. F. Schneider, "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial," *ACM Computing Surveys*, Vol. 22, No. 4, pp. 299–319, 1990
14. Secure Computing Corporation, "Intrusion Tolerant Server Infrastructure," *DARPA OASIS Program*, http://www.tolerantsystems.org/ProjectSummaries/Intrusion_Tolerant_Server_Infrastructure.html
15. S. Singh, M. Cukier, and W. H. Sanders, "Probabilistic Validation of an Intrusion-Tolerant Replication System," *Proc. Intl Conf. on Dependable Sys. and Networking (DSN-2003)*, pp. 615–624, 2003