November 2003

UILU-ENG-03-2227
CRHC-03-13

STOCHASTIC MODELING OF INTRUSION-TOLERANT SERVER ARCHITECTURES FOR DEPENDABILITY AND PERFORMANCE EVALUATION

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

Coordinated Science Laboratory
1308 West Main Street, Urbana, IL 61801
University of Illinois at Urbana-Champaign

SF 298 MASTER COPY KEEP THIS COPY FOR REPRODUCTION PURPOSES

REPORT DOCUMENTATION PAGE

Form Approved OMB NO. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE November 2003	3. REPORT TYPE AND DATES COVERED

4. TITLE AND SUBTITLE Stochastic Modeling of Intrusion-Tolerant Server Architectures for Dependability and Performance Evaluation	5. FUNDING NUMBERS F30602-00-C-0172
6. AUTHOR(S) Vishu Gupta, Vinh Lam, HariGovind V. Ramasamy, William H. Sanders, and Sankalp Singh	
7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(ES) Coordinated Science Laboratory University of Illinois 1308 West Main St. Urbana, IL 61801	8. PERFORMING ORGANIZATION REPORT NUMBER UILU-ENG-03-2227 (CRHC-03-13)
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) DARPA 3701 North Fairfax Drive Arlington, VA 22203-1714	10. SPONSORING / MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES	
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution unlimited.	12 b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

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 describe in detail how the models were constructed using Stochastic Activity Networks (SANs), a variant of stochastic Petri nets. 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 make informed choices for building more secure and survivable server systems.

14. SUBJECT TERMS 1. Intrusion Tolerance; 2. Intrusion-Tolerant Architectures; 3. Security; 4. Intrusion-Tolerant Communication; 5. Stochastic Activity Networks; 6. Stochastic Petri Nets	15. NUMBER IF PAGES 86
	16. PRICE CODE

17. SECURITY CLASSIFICATION OR REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

Stochastic Modeling of Intrusion-Tolerant Server Architectures for Dependability and Performance Evaluation*

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

Coordinated Science Laboratory,
Electrical and Computer Engineering Department, and
Department of Computer Science,
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 describe in detail how the models were constructed using Stochastic Activity Networks (SANs), a variant of stochastic Petri nets. 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.

Keywords: Intrusion Tolerance, Intrusion-Tolerant Architectures, Security, Intrusion-Tolerant Communication, Stochastic Activity Networks, Stochastic Petri Nets

*This research has been supported by DARPA contract F30602-00-C-0172

†Author names appear in alphabetical order. Authors made equal contributions to the research.

1 Introduction

Intrusion tolerance [8] 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 [9], ITSI [16], ITUA [4], and PBFT [3]. 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 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 [13], as shown by Singh et al. [17], 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) [11, 14] 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. Some of the input parameters include intrusion detection rate, repair rate of corrupted servers, single-phase and multi-phase attack rate and probability, and firewall filter rate. We define various measures on the system that characterize system performance and intrusion tolerance. They include *"strong"* and *"weak" unavailability*[1] of the system, *productive* and *unproductive throughput*[1], and *fraction of online servers that are corrupted*[1].

[1]We explain these terms in Section 4.

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. By replicating the servers of the service system, we could potentially improve throughput performance and provide server systems with high availability and scalability. All such systems also have some mechanism by which the incoming requests are spread among the servers. The reader is referred to [2] for a detailed classification of the various approaches for routing the requests among the distributed server nodes.

Since transparency is a design criterion in all the architectures, 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. For instance, if an attacker identifies the specific operating system of a target platform, he/she can focus an attack, minimizing time and attack signatures. Client-based, DNS-based, and server-based routing mechanisms (see [2]) 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. They are

- Centralized Routing Centralized Management (CRCM),
- Multicast Routing Centralized Management (MRCM),
- State Machine Replication (SMR), and
- Multicast Routing Decentralized Management (MRDM).

All four intrusion-tolerant distributed web server architectures that we evaluate have the following components in one form or another:

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

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

3. *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).

4. *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.

5. *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.

6. *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.

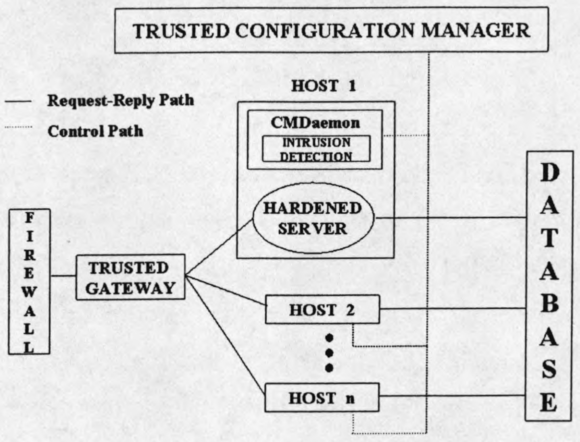
7. *Firewall:* This component filters incoming requests based on certain policies.

8. *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. A comparison of the performance benefits of various client-server database architectures is beyond the scope of this paper. Interested readers are referred to [1, 6, 7].

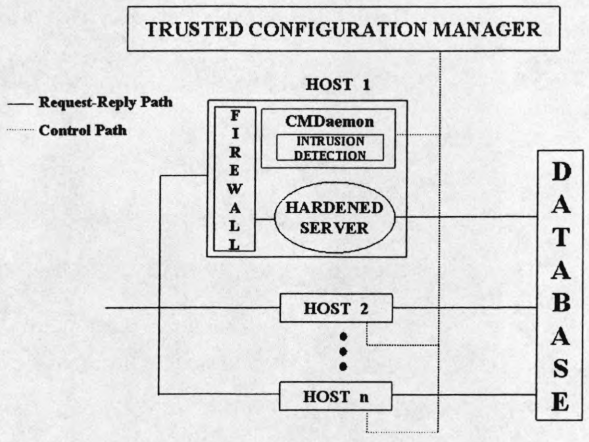
The four architectures differ in how the above components interact with each other, their placement, and which of them are trusted. A "trusted" component is one that is assumed not to fail. We now describe each of the architectures we are considering in more detail.

2.1 Centralized Routing Centralized Management (CRCM)

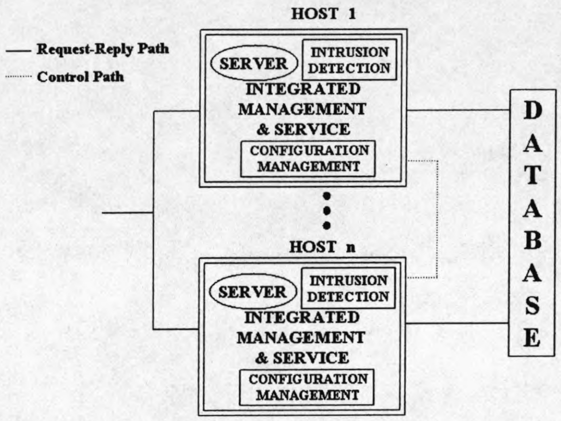
The goal of the CRCM design is to employ a small number of trusted components to protect a large set of servers and databases. In this design, a Firewall component filters the incoming requests, looking for signatures of commonly known attacks. The Gateway is a trusted component. An incoming request passes



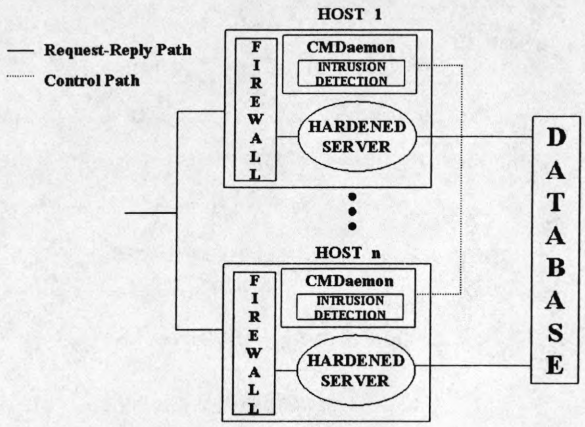
(a) Centralized Routing Centralized Management



(b) Multicast Routing Centralized Management



(c) State Machine Replication



(d) Multicast Routing Decentralized Management

Figure 1: Architecture Block Diagrams

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 might run different 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.

2.2 Multicast Routing Centralized Management (MRCM)

In the MRCM architecture, intrusion tolerance is obtained through hardened, heterogeneous platforms. We achieve this hardening 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 which case, if there are two web servers, the rule might be that server 1 processes a request if the last digit of the client IP address is from 0-4, and that otherwise server 2 processes the request. 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.

2.3 State Machine Replication (SMR)

The SMR design employs a state-machine-replication-based approach [15] that tolerates malicious faults. A replication protocol that tolerates Byzantine faults, similar to the one described in [3], 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 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.

2.4 Multicast Routing Decentralized Management (MRDM)

The MRDM design is a hybrid of the previous 3 architectures, and tries to achieve a tradeoff 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 [12]) that requires the participation of at least $3f + 1$ nodes to tolerate f simultaneous intrusions. 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 the same level of correctness as SMR, 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. Hence, this architecture does not guarantee strict correctness of replies. 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.

2.5 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

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

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

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 Intrusion-Tolerant Server Architectures

3.1 Stochastic Activity Networks

As stated in the introduction, we use SANs, a stochastic extension of Petri nets, as our model representation. Here we present only a brief overview of SANs. Interested readers are urged to consult [11, 14] for additional details on SANs.

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.

3.2 Description of Models

We have modeled the four architectures described in Section 2 as composed stochastic activity networks. Atomic models were built for various components of each architecture, and complete models were then built using replicate and join operations. Replicated and joined sub-models in a composed model can interact with each other through a set of places (called *shared* places) that are common to multiple sub-models.

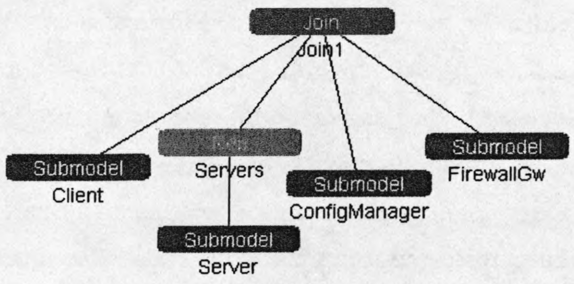
The salient features that we have tried to model for each architecture include generation of client requests and attacks, organization of firewalls and filtering of requests, organization of servers and distribution of requests to servers, servicing of requests and effect of attacks, detection mechanisms, 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.

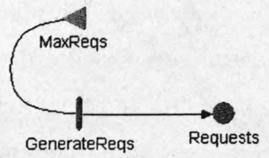
We now provide a description of the models of the individual architectures.

3.2.1 Centralized Routing Centralized Management (CRCM)

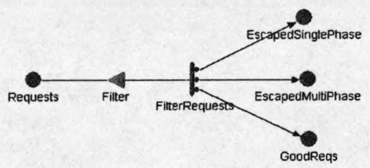
Figure 2(a) shows the composed model for CRCM described in Section 2.1. The model consists of four atomic SAN submodels: **Client**, **Server**, **ConfigManager**, and **FirewallGw**. The **Server** submodel is replicated `NumServers` times, where `NumServers` is a global variable indicating the number of hosts



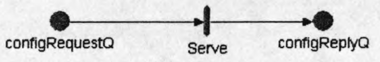
(a) Composed Model



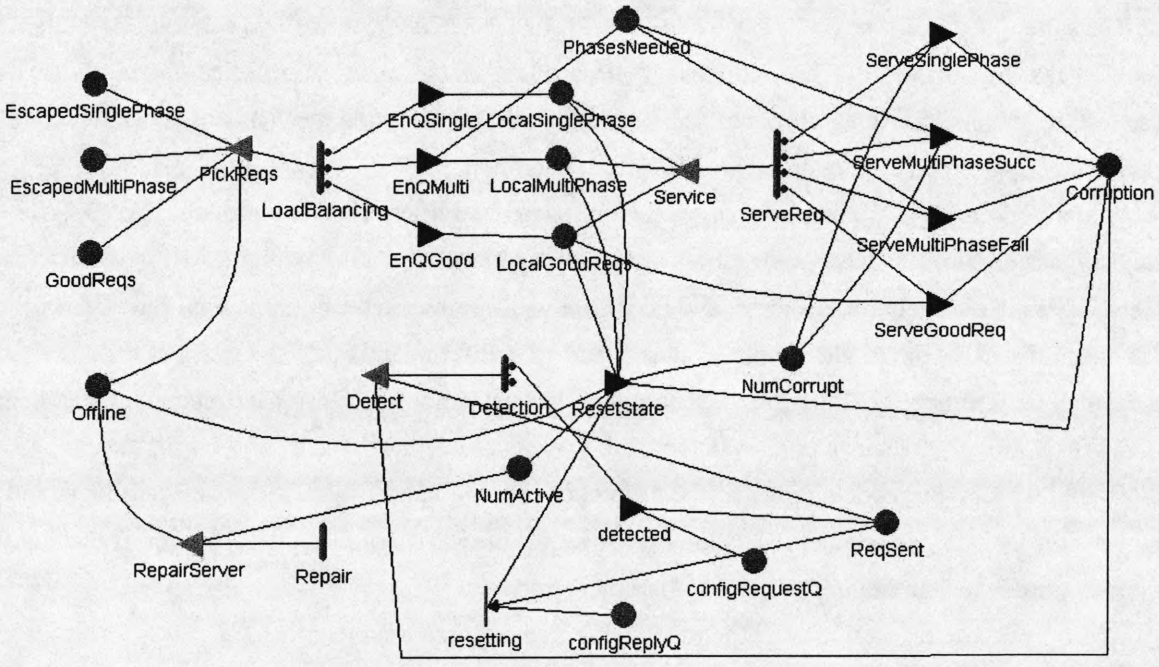
(b) SAN Submodel for Client



(c) SAN Model for Firewall



(e) SAN Submodel for ConfigManager



(d) SAN Submodel for Server

Figure 2: SAN Models for CRCM

running servers.

Figure 2(b) shows the SAN representation of the **Client** submodel. This SAN models the generation of incoming requests to the system from the clients. Since requests have to pass through a firewall and a gateway before they are distributed to individual servers, we have a single unreplicated client submodel in the composed model. The place *Requests* is shared with the **Firewall** submodel, and its marking represents the number of new requests waiting to pass through the firewall. The activity *GenerateReqs* is responsible for generating new requests. The input gate *MaxReqs* allows the activity to be enabled only if the number of waiting requests is lesser than an upper bound, `MaxRequests`. We bound lengths of all places in our model that serve as queues between components so that the model has a finite number of states.

Figure 2(c) shows the SAN representation of the **FirewallGw** submodel. This SAN models the firewall that filters incoming requests with known attack signatures. The place *Requests* is shared with the **Client** submodel, and keeps track of the number of requests waiting to be processed. Upon firing, the activity *FilterRequests* consumes a token from *Requests*, and, according to the probability distribution of its three cases, designates the request to be a good request, a single-phase attack, or a multi-phase attack (the number of phases in a multi-phase attack is set in the **Server** submodels). A case corresponding to an attack is chosen with the probability that such an attack will occur and will escape the firewall. The output places are shared with the **Server** submodels. A point to note here is that since all packets pass through the firewall to reach the servers, 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, and since effects of both single-phase and multi-phase attacks are similar (they result in corruption of a server).

Figure 2(d) shows the SAN representation of a **Server** submodel. This SAN models the 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.

As described in Section 2.1, the centralized load-balancing gateway randomly forwards each incoming request to the system to one of the active servers. The activity *LoadBalancing* is present in each **Server** submodel to model this distribution. The places representing local queues are introduced to accurately model the distribution scheme, which is different from an idealized scheme in which a server picks a request from the global pool as soon as it becomes idle. If the case corresponding to a multi-phase attack is chosen, *LocalMultiPhase* is set to a random integer between 2 and `MaxPhases` (a tunable global variable representing the maximum number of phases in a multi-phase attack), and *PhasesNeeded*, which represents the number of phases that need to be successful for the multi-phase attack to be successful, is set to a fixed fraction of *LocalMultiPhase*.

The activity *Service* represents the servicing of requests in the server's local queue, as well as the effect of attacks on the server. 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. If the number of phases represented by the marking of *PhasesNeeded* are successful on this server, the value of *Corruption* is set to `MaxPhases`, indicating the successful completion of the multi-phase attack. On the other hand, if the required number of phases have not completed when the last token is removed from *LocalMultiPhase*, the attack is deemed to be unsuccessful, and the marking of *Corruption* is reset to 0. The marking of the shared place *NumCorrupt* (which represents the number of corrupt but as-yet-undetected online servers in the system) is incremented upon completion of a successful attack on the server.

The activity *Detection* represents the detection component present on the server host. If the case corresponding to a successful detection is chosen, the output gate *ResetState* is responsible for sending the alert information to the Configuration Manager. Upon receipt of a response from the Manager, the server is taken offline (through setting of the marking of local place *Offline* to 1), thus reducing the number of active servers (represented by the shared place *NumActive*), decrementing the number of corrupt servers if the server represented by this SAN was completely corrupt at the time of detection, and purging all the requests currently in the server's local queues. A point to note is that *Detection* is enabled if the marking of *Corruption* is nonzero, and 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).

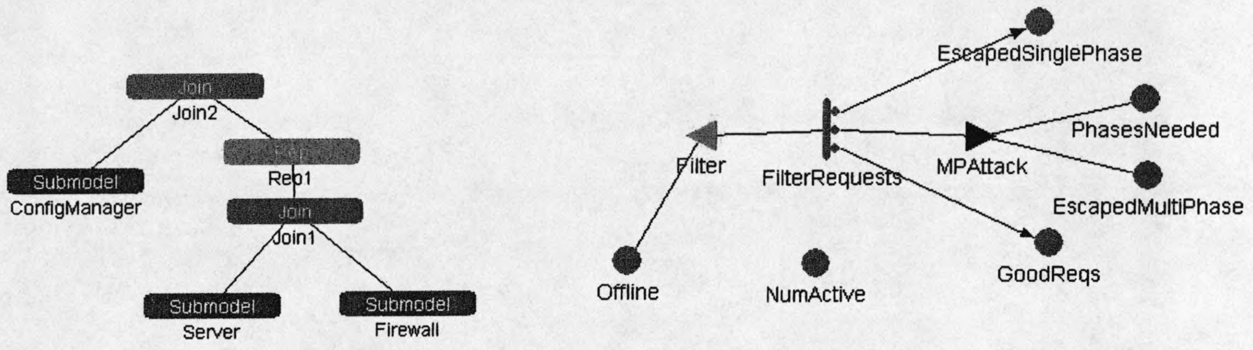
Once a server is taken offline, the Configuration Manager informs the load-balancing gateway about this change, and the latter no longer forwards new requests to the server. We model that by enabling the activity *LoadBalancing* only when the marking of *Offline* is 0.

The activity *Repair* represents the process of reinitializing the state of the server. Upon firing, it increments the number of active servers in the system, and sets the marking of *Offline* to 0, allowing the server to receive requests again.

3.2.2 Multicast Routing Centralized Management (MRCM)

Figure 3.2.2 shows the composed model for MRCM, which was described in Section 2.2. It consists of three atomic SAN models: **Server**, **Firewall**, and **ConfigManager**. Since a firewall is now present on each host running the server, the submodels are joined to form a model of each host (*Join1*). The resulting submodel is replicated `NumServers` times to form a model of the set of servers.

Figure 3.2.2 shows the SAN representation of the **Firewall** submodel. This SAN models the generation of requests from the client, distribution of requests to the servers, filtering of requests as they pass through the firewall, and generation of multiple phases of a multi-phase attack. As described in Section 2.2, in this



(a) Composed Model (b) SAN Submodel for Firewall

Figure 3: SAN Models for MRCM

architecture, each request goes to all the servers, and exactly one of them picks it up (using a deterministic function) for service, while the other servers discard it. We model this by generating requests for each server separately, not through centralized request generation as we did for CRCM. To optimize the model's state-space, we do not model the request generation in a separate client model, since the client submodel would have to be replicated with each server. 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.

The SAN representation for the **Server** submodel is similar to the SAN for the **Server** submodel of CRCM described above. The significant differences are as follows. There are no global request queues (since requests are generated for each server); the queues are simply shared with the **Firewall** submodel. 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*.

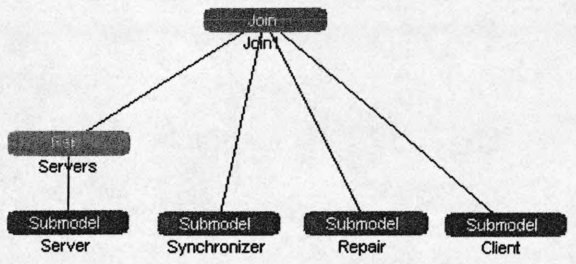
The SAN representation for the **ConfigManager** submodel is identical to the SAN for the **ConfigManager** submodel of CRCM.

3.2.3 State Machine Replication (SMR)

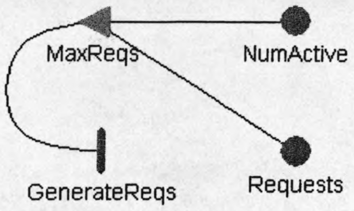
Figure 4(a) shows the composed model for SMR, which was described in Section 2.3. The model consists of four atomic SAN submodels: **Client**, **Server**, **Synchronizer**, and **Repair**. The **Server** submodel is replicated `NumServers` times to form the set of *Servers*.

Figure 4(b) shows the SAN representation of the **Client** submodel. This SAN models the generation of incoming requests to the system. Since each request is sent to all the active servers, we have centralized request generation, and upon firing of the activity *GenerateReqs*, the marking of the shared place *Requests* is increased by the marking of *NumActive* (i.e., we send a copy of the request to each active server).

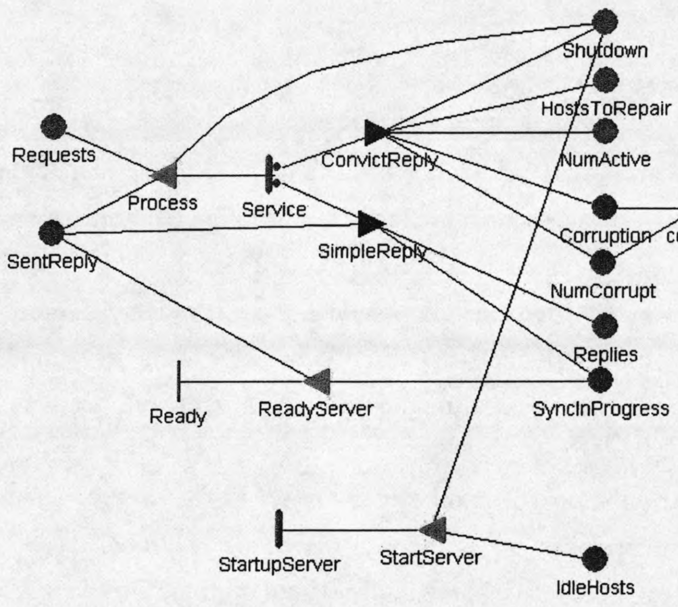
Figure 4(c) shows the SAN representation of the **Server** submodel. This SAN models the processing of client requests by a server, attacks on a server, performance of Byzantine agreement between servers before a



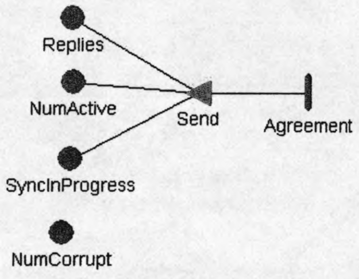
(a) Composed Model



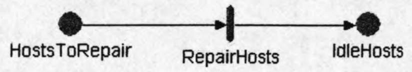
(b) SAN Submodel for Client



(c) SAN Submodel for Server



(d) SAN Model for Synchronizer



(e) SAN Submodel for Repair

Figure 4: SAN models for SMR

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.

The activity *Attack* represents the attacks on the server. Since the system reacts identically to single- 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 firewall, 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.

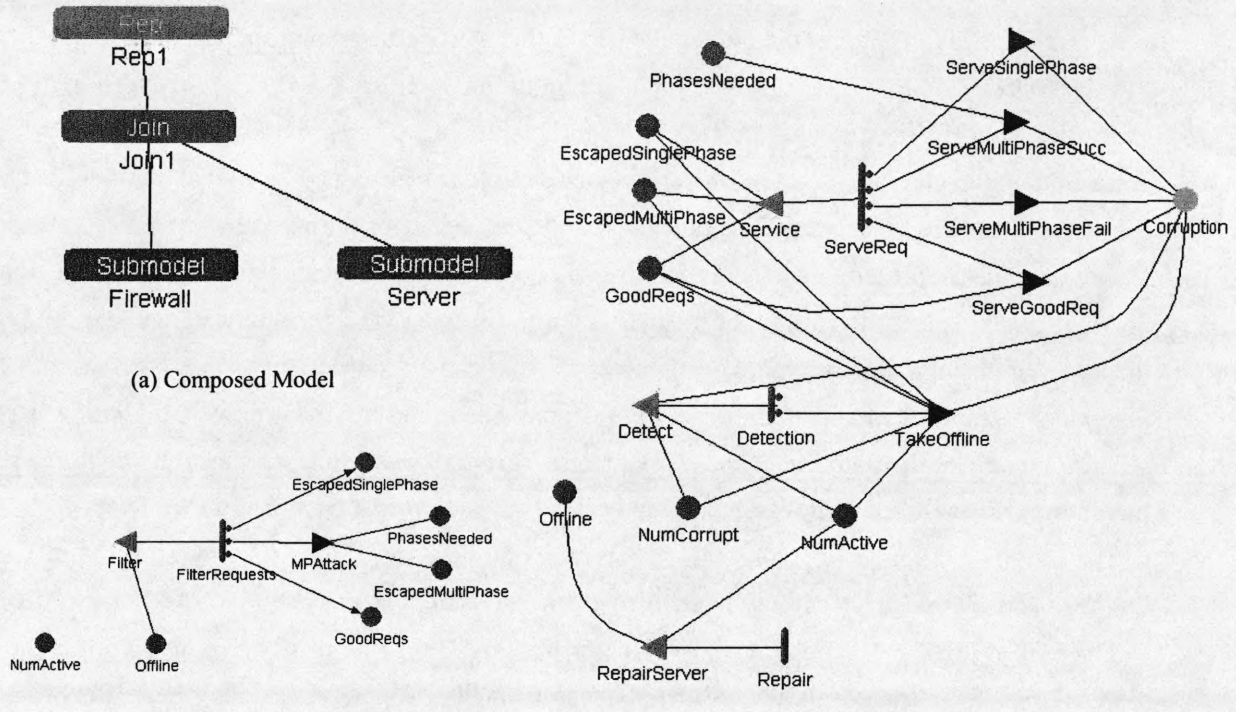
The activity *Service* represents the processing of a client request by the server, and the reaching of Byzantine agreement among the servers on the reply. The probability distribution of the two cases is governed by the marking of the local place *Corruption*. If the marking of *Corruption* is 0, the case corresponding to the output gate *SimpleReply* is chosen with a probability of 1. *SimpleReply* models the sending of a correct reply by the server. It increments the marking of *Replies*, and puts a token in the shared place *SyncInProgress* if its marking was 0. Hence, the first active server that starts processing of a particular request puts a 1 in *SyncInProgress*. *SimpleReply* also sets the marking of the local place *SentReply* to 1. The activity *Service* is enabled only if *SentReply* has no tokens. 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. *ConvictReply* first checks if there are enough uncorrupted hosts to reach a Byzantine agreement, i.e., the marking of *NumCorrupt* is less than a third of the marking of *NumActive*. If a Byzantine agreement can be reached, the marking of the local place *Shutdown* is set to 1 (indicating that this server has been taken offline), the markings of *NumActive* and *NumCorrupt* are decremented, and the marking of *Corruption* is reset to 0. We also increment the marking of the shared place *HostsToRepair*, 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.

The activity *StartupServer* represents the starting of a new server on a standby host, to replace one that has been shut down. This activity is enabled if the server represented by this SAN has been shut down, and there is at least one standby host. When *StartupServer* fires, the marking of *Shutdown* is set to 0, and the marking of the shared place *IdleHosts* is decremented. 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 instantaneous activity *Ready* is enabled if the server has sent a reply (a token is present in *SentReply*) and the marking of *SyncInProgress* is 0. This implies that all active servers have sent their replies, and hence a reply has been sent to the client, and the servers are ready to process a new request. When *Ready* fires, the marking of *SentReply* is reset to 0. That achieves a kind of "barrier synchronization," so that all servers start on a new request only after all of them have completed the processing of the previous request. This models the requirement of maintaining consistent state across all correct replicas.

Figure 4(d) shows the SAN representation of the **Synchronizer** submodel. This SAN models the completion of the response to a client request. The activity *Agreement* is enabled if the markings of *Replies* and *NumActive* are identical, i.e., each online server has sent a reply to the current request. The delay at this activity increases with the number of servers in the system. However, instead of increasing linearly, it increases as a step function, with jumps whenever the number of servers is of the form $3f + 1$ (i.e., it jumps at 4, 7, 10, and so on). When *Agreement* fires, it resets the markings of *Replies* and *SyncInProgress* to 0 so that all of the servers can work on the next request. The **Synchronizer** submodel is needed since the servers have to maintain the same state, because they use state-machine replication; therefore, a server cannot start working on a new request before the current request has been completely handled.

Figure 4(e) shows the SAN representation of the **Repair** submodel. This SAN models the repair process of the excluded hosts, which results in their transition to the standby state.



(a) Composed Model

(b) SAN Model for Firewall

(c) SAN Submodel for Server

Figure 5: SAN Models for MRDM

3.2.4 Multicast Routing Decentralized Management (MRDM)

The composed model and atomic SAN submodels for the MRDM architecture are shown in Figure 5. They are quite similar to those for MRCM (Section 3.2.2). 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 the Byzantine agreement algorithm; 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 multi-phase 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 [5] tool to build the SANs, define performance and intrusion tolerance measures, and design studies on the models. We also used Möbius to simulate the models and obtain values for the measures defined on various studies. We defined several measures on each model for use in the studies. They included:

- *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 serve, 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: This measure also characterizes the fraction of time the service was *improper* in the given time interval, but it uses 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.

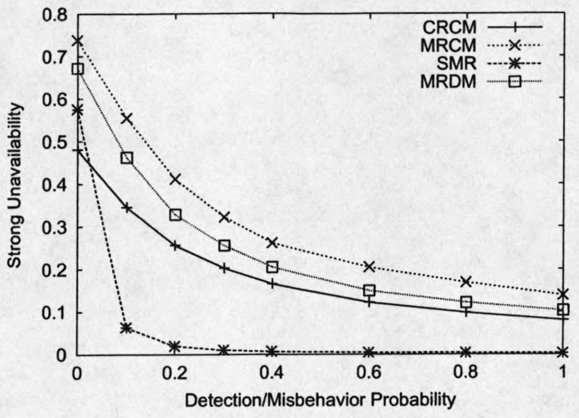
The time interval considered is [0, 30 minutes]. The fraction of corrupt servers is measured at the end of this interval.

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 *mis*behavior during the agreement, by deviating from the protocol specification. That is modeled by the probability of misbehavior.

To see the effect of the probability of detection in the performance and availability of the system, we conducted experiments in which 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). A model for SMR with low misbehavior probability represents an attack scenario in which servers behave correctly (to avoid detection) for a long time after they have been corrupted, and start misbehaving in a correlated manner once enough servers have been compromised to thwart the Byzantine agreement.


(a) Strong Unavailability

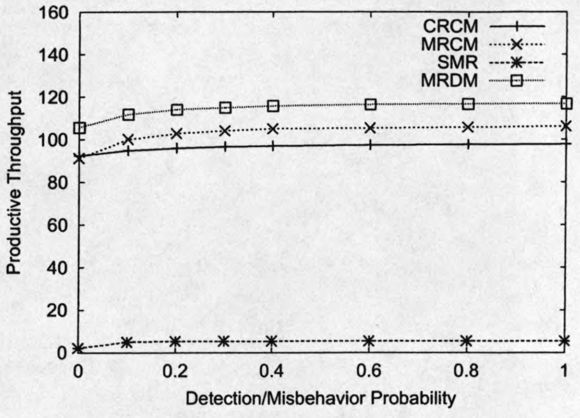
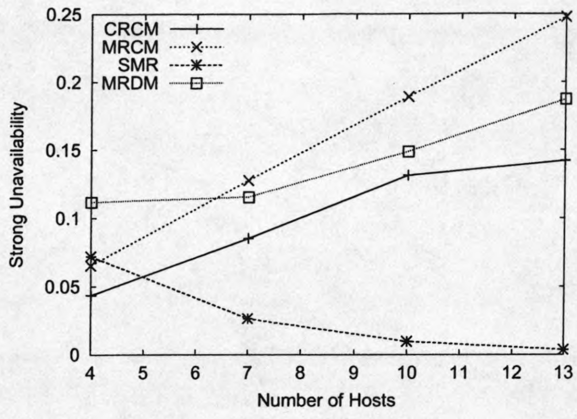

(b) Productive Throughput

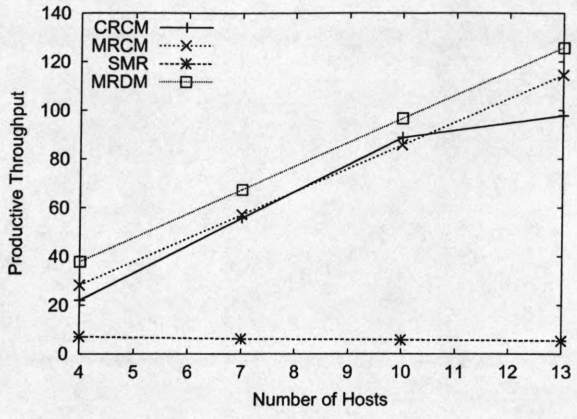
Figure 6: Varying Detection/Misbehavior Probability

As we can see in Figure 6(a), in the absence of an intrusion detection mechanism (or equivalently, in the 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 SMR is initially very sensitive to any increase in 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 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). For that reason, for any probability higher than 0.2, SMR with 12 servers shows almost 0 unavailability.

Figure 6(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 reflect the fact that the number of hosts actually serving client requests varies across these architectures (MRDM has two more hosts than CRCM, and one more host than MRCM).



(a) Strong Unavailability



(b) Productive Throughput

Figure 7: Varying Number of Hosts

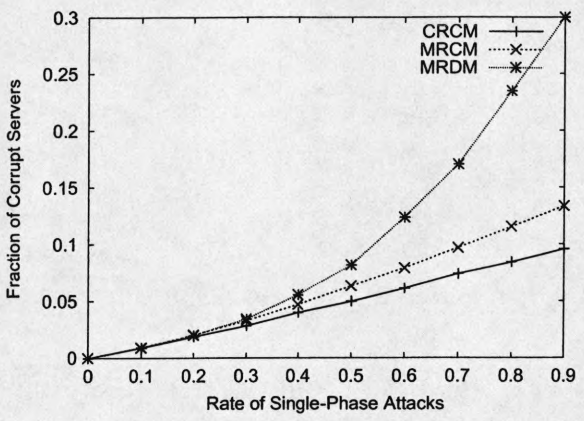
4.2 Comparison under Varying Numbers of Hosts in the System

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.

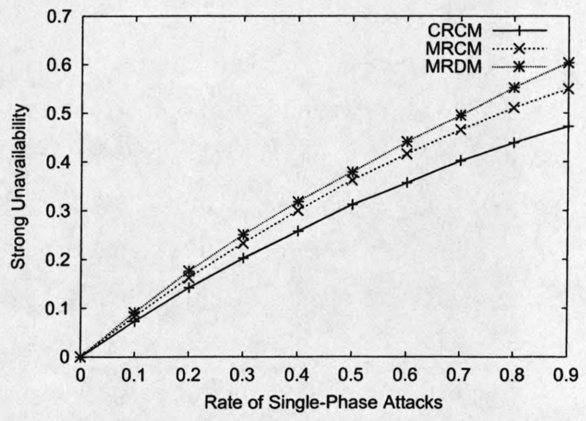
For 4 hosts, SMR and MRDM are more unavailable than CRCM and MRCM (see Figure 7(a)), because they require Byzantine agreement in order to exclude corrupt servers, and 4 servers can tolerate at most one corruption. Given enough time, it may be easy to corrupt one server, and beyond that point, no further 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 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, because of the definition of strong unavailability, 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.

Another interesting point to note is that CRCM does not show an appreciable increase in unavailability above 10 hosts. The reason is that for a fixed 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.

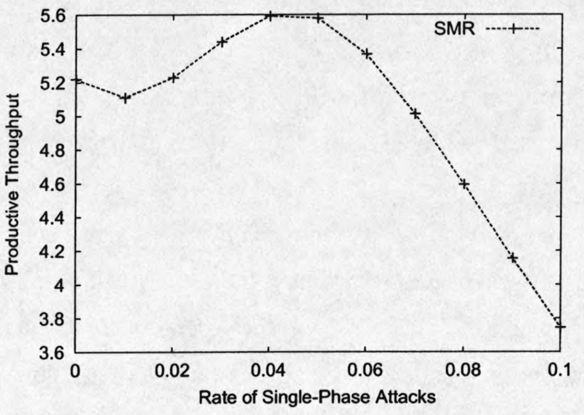
Figure 7(b) shows the productive throughput shown by different architectures as we increase the number of hosts. In SMR, all hosts process every request, so increasing the number of hosts does not help in increasing throughput; rather, productive throughput actually falls a little, because of an increase in agree-



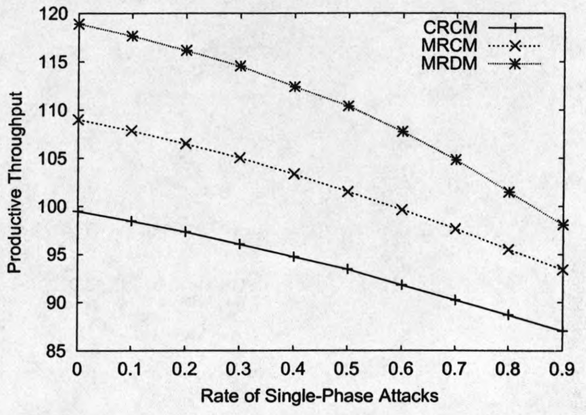
(a) Fraction of Corrupt Servers



(b) Strong Unavailability



(c) Productive Throughput for SMR



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

Figure 8: Variation in Measures with Varying Single-phase Attack Probability

ment 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 any 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

We conducted studies on the models to observe the effect of varying single-phase attack rates on the performance and intrusion-tolerance characteristics of the architectures. For the CRCM, MRCM, and MRDM architectures, the probability that an incoming request is a single-phase attack was varied from 0 to 0.009 in increments of 0.001 (which results in attack rates varying from 0 to 0.9 in increments of 0.1, since the request arrival rate is 100). Since we were looking at response to single-phase attacks, the probability of multi-phase attacks was set to 0. For SMR, the attack rate was varied along the same lines. All the other parameters were the same as we described at the beginning of this section.

Figure 8(a) shows the variation in the fraction of active servers that are corrupt for the CRCM, MRCM, and MRDM architectures. We observe that 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.

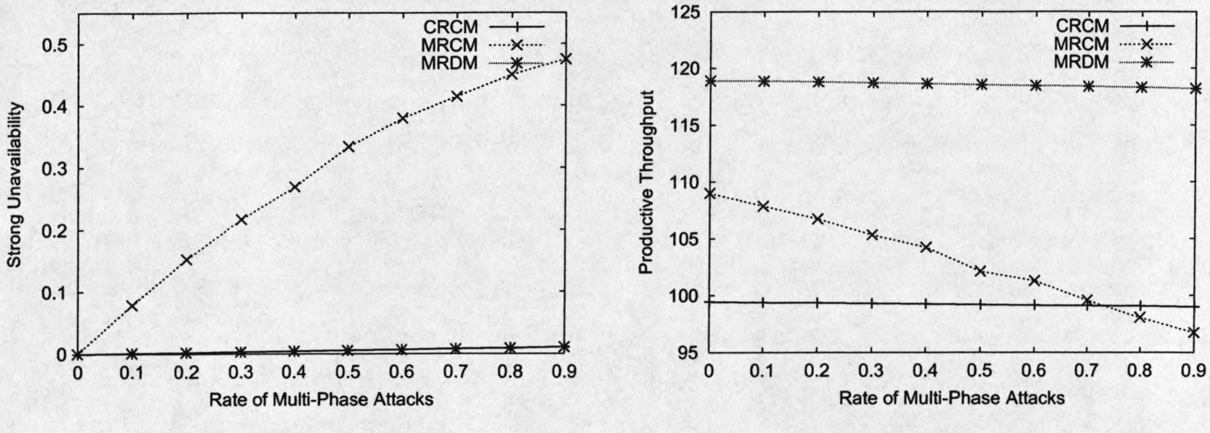
Figure 8(b) shows the variation in strong unavailability for the CRCM, MRCM, and MRDM architectures. All the architectures perform similarly and are strongly affected by the rate of attacks. CRCM is slightly better due to its strong centralized firewall, and MRDM is slightly worse due to the failure of the Byzantine agreement algorithm for higher attack rates.

Figure 8(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 [12]. Since the throughput varies 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 8(d) shows the variation in productive throughput for the CRCM, MRCM, and MRDM architectures. As expected, 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.

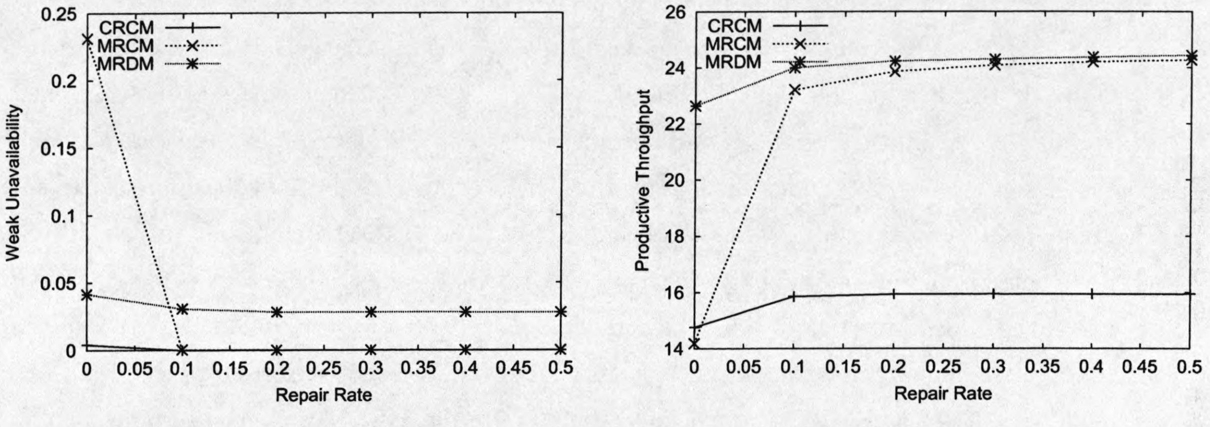
4.4 Comparison under Varying Multi-phase Attack Rates

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 9(a) shows that CRCM and MRDM (coinciding lines) perform better than MRCM with respect to strong unavailability. The reason is that multi-



(a) Strong Unavailability (b) Productive Throughput

Figure 9: Variation in Measures with Varying Multi-phase Attack Probability



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

Figure 10: Variation in Measures with Varying Repair Rates

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 9(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.

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 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. Only if repairs take less time than the duration between successive attacks is it possible to provide continuous service even in the presence of successful attacks. Thus, we can intuitively predict that a faster repair rate is crucial for ensuring that the system provides continuous service.

Figure 10 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, the graphs show that the good 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 the negative performance variables (such as 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 10(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 10(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 [12].

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 perform better in terms of availability than architectures with no trusted components when the level of redundancy in the system is not very large. However, practical experience [10] 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 usually come at the expense of 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 request dispersion mechanisms that introduce unpredictability in request routing are highly effective in defense against multi-phase attacks. It is critical that the mean time to repair be much less than the mean time between attacks. However, beyond a certain point, increasing the repair rate does not give appreciable added benefits.

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: The authors thank other members of the ITUA team for their helpful comments, Dr. Marinho Barcellos for his comments and help in improving the manuscript, and Jenny Applequist for her editorial assistance.

References

[1] B. Cahoon, K. S. McKinley, and Z. Lu, "Evaluating the Performance of Distributed Architectures for Information Retrieval using a Variety of Workloads," *IEEE Trans. on Information Sys.*, Vol. 18, No. 1, pp. 1–43, 1997.

[2] 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.

[3] M. Castro and B. Liskov, "Practical Byzantine Fault Tolerance," *Proc. Third Symp. on Operating Sys. Design and Implementation (OSDI '99)*, pp. 173–186, 1999.

[4] 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.

[5] 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.

[6] A. Delis and N. Roussopoulos, "Performance Comparison of Three Modern DBMS Architectures," *IEEE Transactions on Software Engineering*, Vol. 19, No. 2, pp. 120–138, 1993.

[7] 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.

[8] Y. Deswarte, L. Blain, J. C. Fabre, "Intrusion Tolerance in Distributed Computing Systems," *Proc. IEEE Symposium on Security and Privacy*, pp. 110–121, 1991.

[9] 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.

[10] 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.

[11] J. F. Meyer, A. Movaghar, and W. H. Sanders, "Stochastic Activity Networks: Structure, Behavior, and Application," *Proc. Intl Workshop on Timed Petri Nets*, pp. 106–115, 1985.

[12] 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)*, Washington, DC, pp. 229–238, 2002.

[13] 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.

[14] 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.

[15] F. Schneider, "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial," *ACM Computing Surveys*, Vol. 22, No. 4, pp. 299–319, 1990.

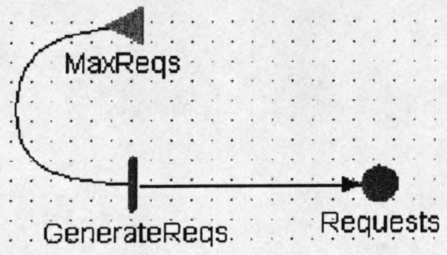
[16] Secure Computing Corporation, "Intrusion Tolerant Server Infrastructure," DARPA OASIS Program, http://www.tolerantsystems.org/ProjectSummaries/Intrusion_Tolerant_Server_Infrastructure.html.

[17] 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.

APPENDIX A.1

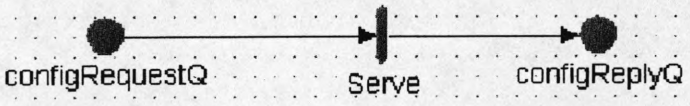
SAN Modeling of the Centralized Routing Centralized Management (CRCM) Architecture

Model: Client



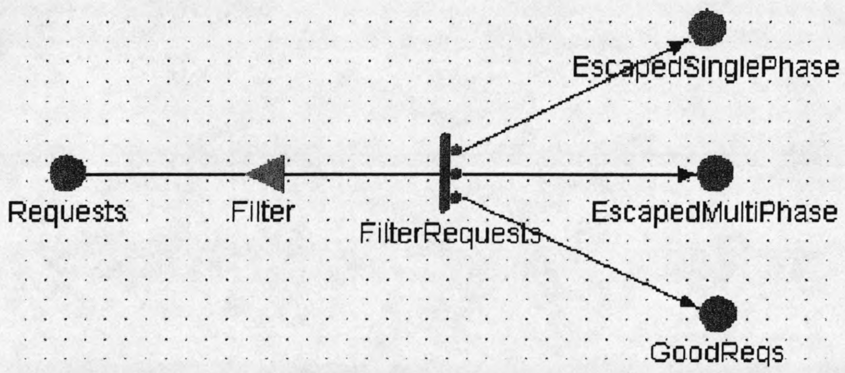
Place Names	Initial Markings
Requests	0
Timed Activity:	**GenerateReqs**
Exponential Distribution Parameters	**Rate** `GEN_REQ_RATE`
Activation Predicate	(none)
Reactivation Predicate	(none)
Input Gate:	**MaxReqs**
Predicate	`(Requests->Mark()) < MAX_REQS`
Function	`;`

Model: ConfigManager



Place Names	Initial Markings
configReplyQ	0
configRequestQ	0
Timed Activity:	**Serve**
Exponential Distribution Parameters	**Rate** `ConfMgrRate`
Activation Predicate	(none)
Reactivation Predicate	(none)

Model: FirewallGateway

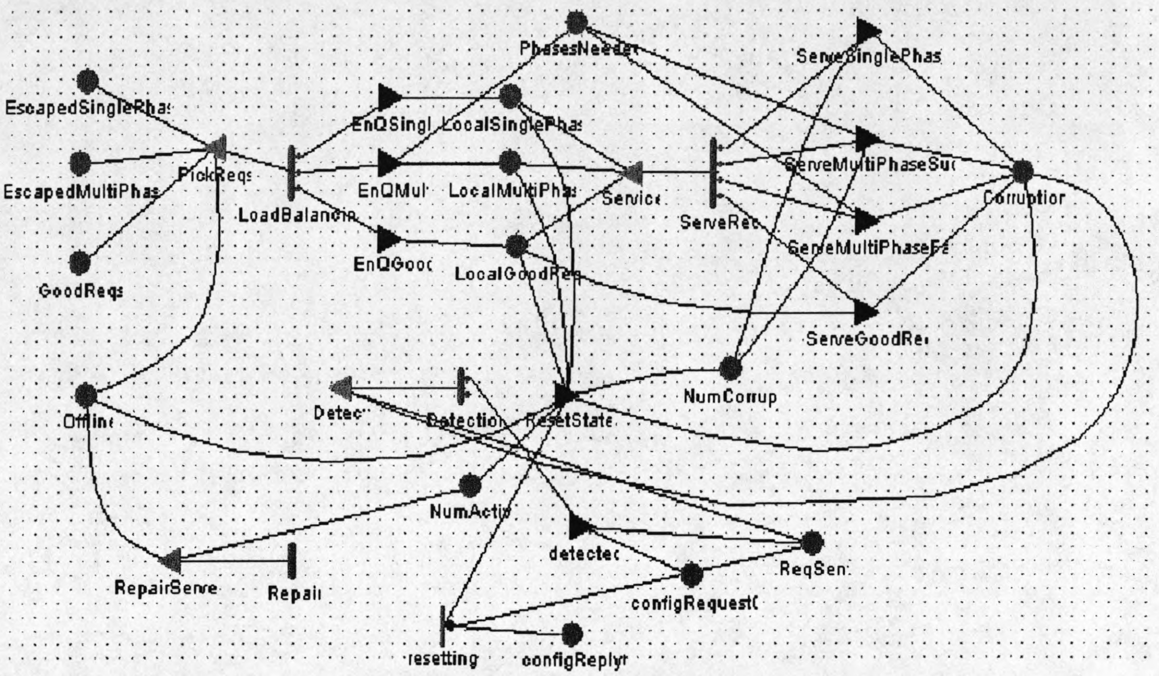


Place Names	Initial Markings
EscapedMultiPhase	0
EscapedSinglePhase	0
GoodReqs	0
Requests	0

Timed Activity:	FilterRequests
Exponential Distribution Parameters	**Rate** `FilterRate`
Activation Predicate	`1`
Reactivation Predicate	`1`
Case Distributions	**case 1** `ProbFiltering*ProbSinglePhase` **case 2** `ProbFiltering * ProbMultiPhase` **case 3** `1 - (ProbSinglePhase + ProbMultiPhase)`

Input Gate:	Filter
Predicate	`Requests->Mark() && (EscapedSinglePhase->Mark() + EscapedMultiPhase->Mark() + GoodReqs->Mark()) < MaxGwQLen`
Function	`Requests->Mark()--;`

Model: Server



Place Names	Initial Markings
Corruption	0
EscapedMultiPhase	0
EscapedSinglePhase	0
GoodReqs	0
LocalGoodReqs	0
LocalMultiPhase	0
LocalSinglePhase	0
NumActive	NumServers
NumCorrupt	0
Offline	0
PhasesNeeded	0
ReqSent	0
configReplyQ	0
configRequestQ	0

Timed Activity:	Detection
Exponential Distribution Parameters	Rate DetectionRate
Activation Predicate	1
Reactivation Predicate	

	1

Case Distributions	**case 1**
	
	`(ProbDetection*Corruption->Mark())/MaxPhases`
	case 2
	
	`1-((ProbDetection*Corruption->Mark())/MaxPhases)`

Timed Activity:	**LoadBalancing**
Exponential Distribution Parameters	**Rate** `LoadBalancingRate`
Activation Predicate	`1`
Reactivation Predicate	`1`

Case Distributions	**case 1**
	
	`(EscapedSinglePhase->Mark()*1.0)/(EscapedSinglePhase->Mark() + EscapedMultiPhase->Mark() + GoodReqs->Mark())`
	case 2
	
	`(EscapedMultiPhase->Mark()*1.0)/(EscapedSinglePhase->Mark() + EscapedMultiPhase->Mark() + GoodReqs->Mark())`
	case 3
	
	`(GoodReqs->Mark()*1.0)/(EscapedSinglePhase->Mark() + EscapedMultiPhase->Mark() + GoodReqs->Mark())`

Timed Activity:	**Repair**
Exponential Distribution Parameters	**Rate** `RepairRate`
Activation Predicate	(none)
Reactivation Predicate	(none)

Timed Activity:	**ServeReq**
Exponential Distribution Parameters	**Rate** `ServiceRate`
Activation Predicate	`1`
Reactivation Predicate	`1`

Case Distributions	**case 1**
	
	`(LocalSinglePhase->Mark()*1.0)/(LocalSinglePhase->Mark() + LocalMultiPhase->Mark() + LocalGoodReqs->Mark())`
	case 2
	
	`(LocalMultiPhase->Mark()*(NumActive->Mark()?(1.0/NumActive->Mark()):0))/(LocalSinglePhase->Mark() + LocalMultiPhase->Mark() + LocalGoodReqs->Mark())`
	case 3

	(LocalMultiPhase->Mark()*(1-(NumActive-
	>Mark()?(1.0/NumActive->Mark()):0)))/(LocalSinglePhase-
	>Mark() + LocalMultiPhase->Mark() + LocalGoodReqs->Mark())
	case 4
	
	(LocalGoodReqs->Mark()*1.0)/(LocalSinglePhase->Mark() +
	LocalMultiPhase->Mark() + LocalGoodReqs->Mark())

Instantaneous Activities Without Cases:

resetting

Input Gate:	**Detect**
Predicate	Corruption->Mark() > 0 && Offline->Mark()==0 && ReqSent->Mark()==0
Function	;

Input Gate:	**PickReqs**
Predicate	(EscapedSinglePhase->Mark() + EscapedMultiPhase->Mark() + GoodReqs->Mark()) > 0 && Offline->Mark()==0 && (LocalSinglePhase->Mark() + LocalGoodReqs->Mark()) < MaxServerQLen
Function	;

Input Gate:	**RepairServer**
Predicate	Offline->Mark()

Function	Offline->Mark()=0;
	NumActive->Mark()++;

Input Gate:	**Service**
Predicate	(LocalSinglePhase->Mark() + LocalMultiPhase->Mark() + LocalGoodReqs->Mark()) > 0
Function	;

Output Gate:	**EnQGood**

Function	if(GoodReqs->Mark()) {
	        GoodReqs->Mark()--;
	         LocalGoodReqs->Mark()++;
	}

Output Gate:	**EnQMulti**

Function	if(EscapedMultiPhase->Mark()) {
	        EscapedMultiPhase->Mark()--;
	        short x = MaxPhases>2?((short)(rand()%(MaxPhases-
	2))+2):2; //uncomment this

```
        //short x=1;   //comment this
        if(LocalMultiPhase->Mark()==0) {
                LocalMultiPhase->Mark() = x;
        }
        else {
                // prob of choosing num phases inversely
prop to # phases
                short y = x + LocalMultiPhase->Mark();
                short z = 1 + (rand()%(y-1)); //uncomment
this
                //short z=1; //comment this
                if(z > x) {
                        LocalMultiPhase->Mark() = x;
                }
        }
        PhasesNeeded->Mark() = 1+(short)(LocalMultiPhase-
>Mark() * fracPhases);
}
```

Output Gate:	**EnQSingle**

Function

```
if(EscapedSinglePhase->Mark()) {
        EscapedSinglePhase->Mark()--;
        LocalSinglePhase->Mark()++;
}
```

Output Gate:	**ResetState**

Function

```
Offline->Mark()=1;
LocalSinglePhase->Mark()=0;
LocalMultiPhase->Mark()=0;
LocalGoodReqs->Mark()=0;
NumActive->Mark()--;
if (Corruption->Mark() == MaxPhases)
        NumCorrupt->Mark()--;
Corruption->Mark()=0;
```

Output Gate:	**ServeGoodReq**

Function

```
if(LocalGoodReqs->Mark()) {
        LocalGoodReqs->Mark()--;
}
```

Output Gate:	**ServeMultiPhaseFail**

Function

```
if(LocalMultiPhase->Mark()) {
        LocalMultiPhase->Mark()--;
        if(LocalMultiPhase->Mark()==0) {
                if (Corruption->Mark() < MaxPhases) {
                        Corruption->Mark()=0;
                }
                PhasesNeeded->Mark() = 0;
        }
}
```

Output Gate:	**ServeMultiPhaseSucc**
Function	

```
if(LocalMultiPhase->Mark()) {
        LocalMultiPhase->Mark()--;
        short flag=0;
        if(Corruption->Mark() < PhasesNeeded->Mark()) {
                Corruption->Mark()++;
                flag=1;
        }
        if (Corruption->Mark() >= PhasesNeeded->Mark())
{
                LocalMultiPhase->Mark()=0;
                PhasesNeeded->Mark() = 0;
                if (flag == 1)
                {
                        Corruption->Mark() = MaxPhases;
                        NumCorrupt->Mark()++;
                }
        }
}
```

Output Gate: **ServeSinglePhase**

Function

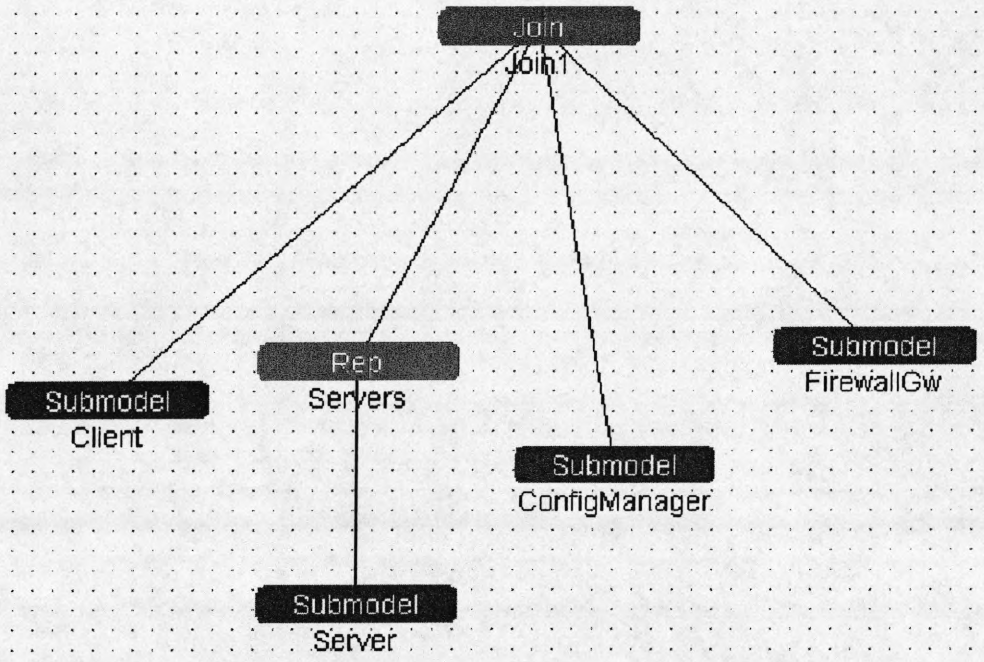
```
if(LocalSinglePhase->Mark()) {
        LocalSinglePhase->Mark()--;
        if (Corruption->Mark() < MaxPhases)
        {
                Corruption->Mark() = MaxPhases;
                NumCorrupt->Mark()++;
        }
}
```

Output Gate: **detected**

Function

```
configRequestQ->Mark() = 1;
ReqSent->Mark() = 1;
```

Model: CRCM



Rep Node	Reps	Shared State Variables
Servers	NumServers	EscapedMultiPhase
		EscapedSinglePhase
		GoodReqs
		NumActive
		NumCorrupt
		configReplyQ
		configRequestQ

Join Node: Join1 :

State Variable Name	Submodel Variables
ConfigReplyQ	Servers->configReplyQ
	ConfigManager->configReplyQ
EscapedMultiPhase	Servers->EscapedMultiPhase
	FirewallGw->EscapedMultiPhase
EscapedSinglePhase	Servers->EscapedSinglePhase
	FirewallGw->EscapedSinglePhase
GoodReqs	Servers->GoodReqs
	FirewallGw->GoodReqs
Requests	Client->Requests
	FirewallGw->Requests
configRequestQ	Servers->configRequestQ
	ConfigManager->configRequestQ

Performance Variable Model: CRCM_PV		
Top Level Model Information	Child Model Name	CRCM
	Model Type	Rep/Join

Performance Variable : throughput			
Affecting Models	Server		
Impulse Functions	Server->ServeReq_case4		
	(Reward is over all Available Models) `return 1;`		
Reward Function	*(Reward is over all Available Models)*		
Simulator Statistics	Type	Time Averaged Interval of Time	
	Options	Estimate Mean	
		Include Lower Bound on Interval Estimate	
		Include Upper Bound on Interval Estimate	
		Estimate out of Range Probabilities	
		Confidence Level is Relative	
	Parameters	Start Time	0.0
		Stop Time	30.0
	Confidence	Confidence Level	0.95
		Confidence Interval	0.1

Performance Variable : arrivalrate			
Affecting Models	FirewallGw		
Impulse Functions	FirewallGw->FilterRequests_case1		
	(Reward is over all Available Models) `return 1;`		
	FirewallGw->FilterRequests_case2		
	(Reward is over all Available Models) `return 1+(MaxPhases/2.0);`		
	FirewallGw->FilterRequests_case3		
	(Reward is over all Available Models) `return 1;`		
Reward Function	*(Reward is over all Available Models)*		
Simulator Statistics	Type	Time Averaged Interval of Time	
	Options	Estimate Mean	
		Include Lower Bound on Interval Estimate	
		Include Upper Bound on Interval Estimate	
		Estimate out of Range Probabilities	
		Confidence Level is Relative	
	Parameters	Start Time	0.0
		Stop Time	30.0

	Confidence	Confidence Level	0.95
		Confidence Interval	0.1

Performance Variable : fracCorruptServers			
Affecting Models	Server		
Impulse Functions			

Reward Function	*(Reward is over all Available Models)*
	`if (Server->Corruption->Mark() == MaxPhases && Server->Offline->Mark() == 0)`
	`    return 1.0/Server->NumActive->Mark();`		

Simulator Statistics	Type	Instant of Time	
	Options	Estimate Mean	
		Include Lower Bound on Interval Estimate	
		Include Upper Bound on Interval Estimate	
		Estimate out of Range Probabilities	
		Confidence Level is Relative	
	Parameters	Start Time	30.0
	Confidence	Confidence Level	0.95
		Confidence Interval	0.1

Performance Variable : StrongUnavailability			
Affecting Models	Server		
Impulse Functions			

Reward Function	*(Reward is over all Available Models)*
	`if(Server->NumActive->Mark()==0) return 1.0/NumServers;`
	`if (Server->Offline->Mark()==0 && Server->Corruption->Mark() == MaxPhases)`
	`    return 1.0/Server->NumCorrupt->Mark();`		

Simulator Statistics	Type	Time Averaged Interval of Time	
	Options	Estimate Mean	
		Include Lower Bound on Interval Estimate	
		Include Upper Bound on Interval Estimate	
		Estimate out of Range Probabilities	
		Confidence Level is Relative	
	Parameters	Start Time	0.0
		Stop Time	30.0
	Confidence	Confidence Level	0.95
		Confidence Interval	0.1

Performance Variable : WeakUnavailability		
Affecting Models	Server	
Impulse Functions		

Reward Function	*(Reward is over all Available Models)*
	`if (Server->NumActive->Mark() == Server->NumCorrupt->Mark())`
	`    return 1.0/NumServers;`	

Simulator Statistics	Type	Time Averaged Interval of Time

	Options	Estimate Mean	
		Include Lower Bound on Interval Estimate	
		Include Upper Bound on Interval Estimate	
		Estimate out of Range Probabilities	
		Confidence Level is Relative	
	Parameters	Start Time	0.0
		Stop Time	30.0
	Confidence	Confidence Level	0.95
		Confidence Interval	0.1

Performance Variable : numCorrupt			
Affecting Models	Server		
Impulse Functions			
Reward Function	*(Reward is over all Available Models)* `return (Server->Corruption->Mark()>=MaxPhases);`		
Simulator Statistics	Type	Instant of Time	
	Options	Estimate Mean	
		Include Lower Bound on Interval Estimate	
		Include Upper Bound on Interval Estimate	
		Estimate out of Range Probabilities	
		Confidence Level is Relative	
	Parameters	Start Time	30.0
	Confidence	Confidence Level	0.95
		Confidence Interval	0.1

Performance Variable : goodthroughput			
Affecting Models	Server		
Impulse Functions	Server->ServeReq_case4		

	(Reward is over all Available Models)
	`if (Server->Corruption->Mark() < MaxPhases)`
	`        return 1;`		

Reward Function	*(Reward is over all Available Models)*		
Simulator Statistics	Type	Time Averaged Interval of Time	
	Options	Estimate Mean	
		Include Lower Bound on Interval Estimate	
		Include Upper Bound on Interval Estimate	
		Estimate out of Range Probabilities	
		Confidence Level is Relative	
	Parameters	Start Time	0.0
		Stop Time	30.0
	Confidence	Confidence Level	0.95
		Confidence Interval	0.1

Performance Variable : badthroughput			
Affecting Models	Server		
Impulse Functions	Server->ServeReq_case4		
	(Reward is over all Available Models) `if(Server->Corruption->Mark() >= MaxPhases)` `return 1.0;`		
Reward Function	*(Reward is over all Available Models)*		
Simulator Statistics	Type	Time Averaged Interval of Time	
	Options	Estimate Mean	
		Include Lower Bound on Interval Estimate	
		Include Upper Bound on Interval Estimate	
		Estimate out of Range Probabilities	
		Confidence Level is Relative	
	Parameters	Start Time	0.0
		Stop Time	30.0
	Confidence	Confidence Level	0.95
		Confidence Interval	0.1

Performance Variable Model: CRCM_PV_SS		
Top Level Model Information	Child Model Name	CRCM
	Model Type	Rep/Join

Performance Variable : SSfracCorruptServers			
Affecting Models	Server		
Impulse Functions			
Reward Function	*(Reward is over all Available Models)* `if (Server->Corruption->Mark() == MaxPhases && Server->Offline->Mark() == 0)` `return 1.0/Server->NumActive->Mark();`		
Simulator Statistics	Type	Steady State	
	Options	Estimate Mean	
		Include Lower Bound on Interval Estimate	
		Include Upper Bound on Interval Estimate	
		Estimate out of Range Probabilities	
		Confidence Level is Relative	
	Parameters	Initial Transient	5.0
		Batch Size	1000.0
	Confidence	Confidence Level	0.95
		Confidence Interval	0.01

Performance Variable : SSStrongUnavailability	
Affecting Models	Server

Impulse Functions			

Reward Function	*(Reward is over all Available Models)*
	`if(Server->NumActive->Mark()==0) return 1.0/NumServers;`
	`if (Server->Offline->Mark()==0 && Server->Corruption->Mark() == MaxPhases)`
	`        return 1.0/Server->NumCorrupt->Mark();`		

Simulator Statistics	Type	Steady State	
	Options	Estimate Mean	
		Include Lower Bound on Interval Estimate	
		Include Upper Bound on Interval Estimate	
		Estimate out of Range Probabilities	
		Confidence Level is Relative	
	Parameters	Initial Transient	5.0
		Batch Size	1000.0
	Confidence	Confidence Level	0.95
		Confidence Interval	0.01

Performance Variable : SSWeakUnavailability			
Affecting Models	Server		
Impulse Functions			

Reward Function	*(Reward is over all Available Models)*
	`if (Server->NumActive->Mark() == Server->NumCorrupt->Mark())`
	`        return 1.0/NumServers;`		

Simulator Statistics	Type	Steady State	
	Options	Estimate Mean	
		Include Lower Bound on Interval Estimate	
		Include Upper Bound on Interval Estimate	
		Estimate out of Range Probabilities	
		Confidence Level is Relative	
	Parameters	Initial Transient	5.0
		Batch Size	1000.0
	Confidence	Confidence Level	0.95
		Confidence Interval	0.01

Performance Variable : SSnumCorrupt		
Affecting Models	Server	
Impulse Functions		
Reward Function	*(Reward is over all Available Models)* `return (Server->Corruption->Mark()>=MaxPhases);`	
Simulator Statistics	Type	Steady State
	Options	Estimate Mean
		Include Lower Bound on Interval Estimate
		Include Upper Bound on Interval Estimate
		Estimate out of Range Probabilities
		Confidence Level is Relative

	Parameters	Initial Transient	5.0
		Batch Size	1000.0
	Confidence	Confidence Level	0.95
		Confidence Interval	0.01

Performance Variable : SSgoodthroughput			
Affecting Models	Server		
Impulse Functions	Server->ServeReq_case4		
	(Reward is over all Available Models)		
	`if (Server->Corruption->Mark() < MaxPhases)` `return 1;`		
Reward Function	*(Reward is over all Available Models)*		
Simulator Statistics	Type	Steady State	
	Options	Estimate Mean	
		Include Lower Bound on Interval Estimate	
		Include Upper Bound on Interval Estimate	
		Estimate out of Range Probabilities	
		Confidence Level is Relative	
	Parameters	Initial Transient	5.0
		Batch Size	1000.0
	Confidence	Confidence Level	0.95
		Confidence Interval	0.01

Performance Variable : SSbadthroughput			
Affecting Models	Server		
Impulse Functions	Server->ServeReq_case4		
	(Reward is over all Available Models)		
	`if(Server->Corruption->Mark() >= MaxPhases)` `return 1.0;`		
Reward Function	*(Reward is over all Available Models)*		
Simulator Statistics	Type	Steady State	
	Options	Estimate Mean	
		Include Lower Bound on Interval Estimate	
		Include Upper Bound on Interval Estimate	
		Estimate out of Range Probabilities	
		Confidence Level is Relative	
	Parameters	Initial Transient	5.0
		Batch Size	1000.0
	Confidence	Confidence Level	0.95
		Confidence Interval	0.01

Range Study Variable Assignments for Study *Detection* in Project *CRCM* :

Variable	Type	Range Type	Range	Increment	Increment Type	Function	n
ConfMgrRate	double	Fixed	10	-	-	-	-
DetectionRate	double	Fixed	.6	-	-	-	-
FilterRate	double	Fixed	200	-	-	-	-
GEN_REQ_RATE	float	Fixed	100	-	-	-	-
LoadBalancingRate	double	Fixed	12	-	-	-	-
MAX_REQS	int	Fixed	50	-	-	-	-
MaxGwQLen	short	Fixed	50	-	-	-	-
MaxPhases	short	Fixed	5	-	-	-	-
MaxServerQLen	short	Fixed	10	-	-	-	-
NumServers	short	Fixed	10	-	-	-	-
ProbDetection	double	Manual	[0, .1, .2, .3, .4, .6, .8, 1]	-	-	-	-
ProbFiltering	double	Fixed	.25	-	-	-	-
ProbMultiPhase	double	Fixed	.001	-	-	-	-
ProbSinglePhase	double	Fixed	.002	-	-	-	-
RepairRate	double	Fixed	0.06	-	-	-	-
ServiceRate	double	Fixed	20	-	-	-	-
fracPhases	double	Fixed	.6	-	-	-	-

Range Study Variable Assignments for Study *CRCM_Study* in Project *CRCM* :

Variable	Type	Range Type	Range	Increment	Increment Type	Function	n
ConfMgrRate	double	Fixed	10	-	-	-	-
DetectionRate	double	Fixed	.6	-	-	-	-
FilterRate	double	Fixed	200	-	-	-	-
GEN_REQ_RATE	float	Fixed	100	-	-	-	-
LoadBalancingRate	double	Fixed	12	-	-	-	-
MAX_REQS	int	Fixed	50	-	-	-	-
MaxGwQLen	short	Fixed	50	-	-	-	-
MaxPhases	short	Fixed	5	-	-	-	-
MaxServerQLen	short	Fixed	10	-	-	-	-
NumServers	short	Fixed	10	-	-	-	-
ProbDetection	double	Fixed	.5	-	-	-	-
ProbFiltering	double	Fixed	.25	-	-	-	-
ProbMultiPhase	double	Fixed	.001	-	-	-	-
ProbSinglePhase	double	Fixed	.002	-	-	-	-
RepairRate	double	Fixed	0.06	-	-	-	-
ServiceRate	double	Fixed	20	-	-	-	-
fracPhases	double	Fixed	.6	-	-	-	-

Range Study Variable Assignments for Study *CRCM_Study_SS* in Project *CRCM* :

Variable	Type	Range Type	Range	Increment	Increment Type	Function	n
ConfMgrRate	double	Fixed	10	-	-	-	-
DetectionRate	double	Fixed	.6	-	-	-	-
FilterRate	double	Fixed	200	-	-	-	-
GEN_REQ_RATE	float	Fixed	100	-	-	-	-
LoadBalancingRate	double	Fixed	12	-	-	-	-
MAX_REQS	int	Fixed	50	-	-	-	-
MaxGwQLen	short	Fixed	50	-	-	-	-
MaxPhases	short	Fixed	5	-	-	-	-
MaxServerQLen	short	Fixed	10	-	-	-	-
NumServers	short	Fixed	10	-	-	-	-
ProbDetection	double	Fixed	.5	-	-	-	-
ProbFiltering	double	Fixed	.25	-	-	-	-
ProbMultiPhase	double	Fixed	.001	-	-	-	-
ProbSinglePhase	double	Fixed	.002	-	-	-	-
RepairRate	double	Fixed	.06	-	-	-	-
ServiceRate	double	Fixed	20	-	-	-	-
fracPhases	double	Fixed	.6	-	-	-	-

Range Study Variable Assignments for Study *MultiPhase* in Project *CRCM* :

Variable	Type	Range Type	Range	Increment	Increment Type	Function	n
ConfMgrRate	double	Fixed	10	-	-	-	-
DetectionRate	double	Fixed	.6	-	-	-	-
FilterRate	double	Fixed	200	-	-	-	-
GEN_REQ_RATE	float	Fixed	100	-	-	-	-
LoadBalancingRate	double	Fixed	12	-	-	-	-
MAX_REQS	int	Fixed	50	-	-	-	-
MaxGwQLen	short	Fixed	50	-	-	-	-
MaxPhases	short	Fixed	5	-	-	-	-
MaxServerQLen	short	Fixed	10	-	-	-	-
NumServers	short	Fixed	10	-	-	-	-
ProbDetection	double	Fixed	.5	-	-	-	-
ProbFiltering	double	Fixed	.25	-	-	-	-
ProbMultiPhase	double	Incremental	[0.0,0.009000000000000001]	.001	Additive	-	-
ProbSinglePhase	double	Fixed	0	-	-	-	-
RepairRate	double	Fixed	0.06	-	-	-	-
ServiceRate	double	Fixed	20	-	-	-	-
fracPhases	double	Fixed	.6	-	-	-	-

Range Study Variable Assignments for Study *NumServers* in Project *CRCM*:

Variable	Type	Range Type	Range	Increment	Increment Type	Function	n
ConfMgrRate	double	Fixed	10	-	-	-	-
DetectionRate	double	Fixed	.6	-	-	-	-
FilterRate	double	Fixed	200	-	-	-	-
GEN_REQ_RATE	float	Fixed	100	-	-	-	-
LoadBalancingRate	double	Fixed	12	-	-	-	-
MAX_REQS	int	Fixed	50	-	-	-	-
MaxGwQLen	short	Fixed	50	-	-	-	-
MaxPhases	short	Fixed	5	-	-	-	-
MaxServerQLen	short	Fixed	10	-	-	-	-
NumServers	short	Incremental	[2,11]	3	Additive	-	-
ProbDetection	double	Fixed	.5	-	-	-	-
ProbFiltering	double	Fixed	.25	-	-	-	-
ProbMultiPhase	double	Fixed	.001	-	-	-	-
ProbSinglePhase	double	Fixed	.002	-	-	-	-
RepairRate	double	Fixed	0.06	-	-	-	-
ServiceRate	double	Fixed	20	-	-	-	-
fracPhases	double	Fixed	.6	-	-	-	-

Range Study Variable Assignments for Study *Repair* in Project *CRCM*:

Variable	Type	Range Type	Range	Increment	Increment Type	Function	n
ConfMgrRate	double	Fixed	10	-	-	-	-
DetectionRate	double	Fixed	.6	-	-	-	-
FilterRate	double	Fixed	200	-	-	-	-
GEN_REQ_RATE	float	Fixed	100	-	-	-	-
LoadBalancingRate	double	Fixed	12	-	-	-	-
MAX_REQS	int	Fixed	50	-	-	-	-
MaxGwQLen	short	Fixed	50	-	-	-	-
MaxPhases	short	Fixed	5	-	-	-	-
MaxServerQLen	short	Fixed	10	-	-	-	-
NumServers	short	Fixed	10	-	-	-	-
ProbDetection	double	Fixed	.5	-	-	-	-
ProbFiltering	double	Fixed	.25	-	-	-	-
ProbMultiPhase	double	Fixed	.001	-	-	-	-
ProbSinglePhase	double	Fixed	.002	-	-	-	-
RepairRate	double	Manual	[0, .01, .02, .03, .04, .05, .1, .15]	-	-	-	-
ServiceRate	double	Fixed	20	-	-	-	-
fracPhases	double	Fixed	.6	-	-	-	-

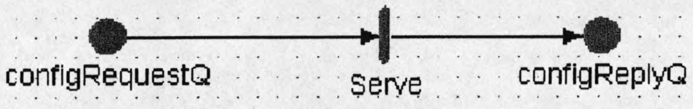
Range Study Variable Assignments for Study *SinglePhase* in Project *CRCM* :

Variable	Type	Range Type	Range	Increment	Increment Type	Function	n
ConfMgrRate	double	Fixed	10	-	-	-	-
DetectionRate	double	Fixed	.6	-	-	-	-
FilterRate	double	Fixed	200	-	-	-	-
GEN_REQ_RATE	float	Fixed	100	-	-	-	-
LoadBalancingRate	double	Fixed	12	-	-	-	-
MAX_REQS	int	Fixed	50	-	-	-	-
MaxGwQLen	short	Fixed	50	-	-	-	-
MaxPhases	short	Fixed	5	-	-	-	-
MaxServerQLen	short	Fixed	10	-	-	-	-
NumServers	short	Fixed	10	-	-	-	-
ProbDetection	double	Fixed	.5	-	-	-	-
ProbFiltering	double	Fixed	.25	-	-	-	-
ProbMultiPhase	double	Fixed	0	-	-	-	-
ProbSinglePhase	double	Incremental	[0.0,0.009000000000000001]	.001	Additive	-	-
RepairRate	double	Fixed	0.06	-	-	-	-
ServiceRate	double	Fixed	20	-	-	-	-
fracPhases	double	Fixed	.6	-	-	-	-

APPENDIX A.2

SAN Modeling of the Multicast Routing Centralized Management (MRCM) Architecture

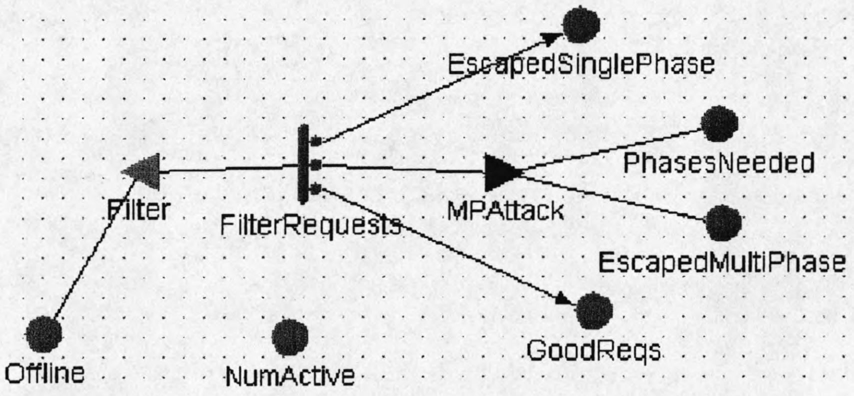
Model: ConfigManager



Place Names	Initial Markings
configReplyQ	0
configRequestQ	0

Timed Activity:	**Serve**
Exponential Distribution Parameters	**Rate** ConfMgrRate
Activation Predicate	(none)
Reactivation Predicate	(none)

Model: Firewall



Place Names	Initial Markings
EscapedMultiPhase	0
EscapedSinglePhase	0
GoodReqs	0
NumActive	NumServers
Offline	0
PhasesNeeded	0

Timed Activity:	**FilterRequests**
Exponential Distribution Parameters	**Rate** `FilterRate*(NumActive->Mark()>0?(NumServers*1.0/NumActive->Mark()):0)`
Activation Predicate	`1`
Reactivation Predicate	`1`
Case Distributions	**case 1** `ProbFiltering*ProbSinglePhase` **case 2** `ProbFiltering*ProbMultiPhase` **case 3** `1 -(ProbSinglePhase+ProbMultiPhase)`

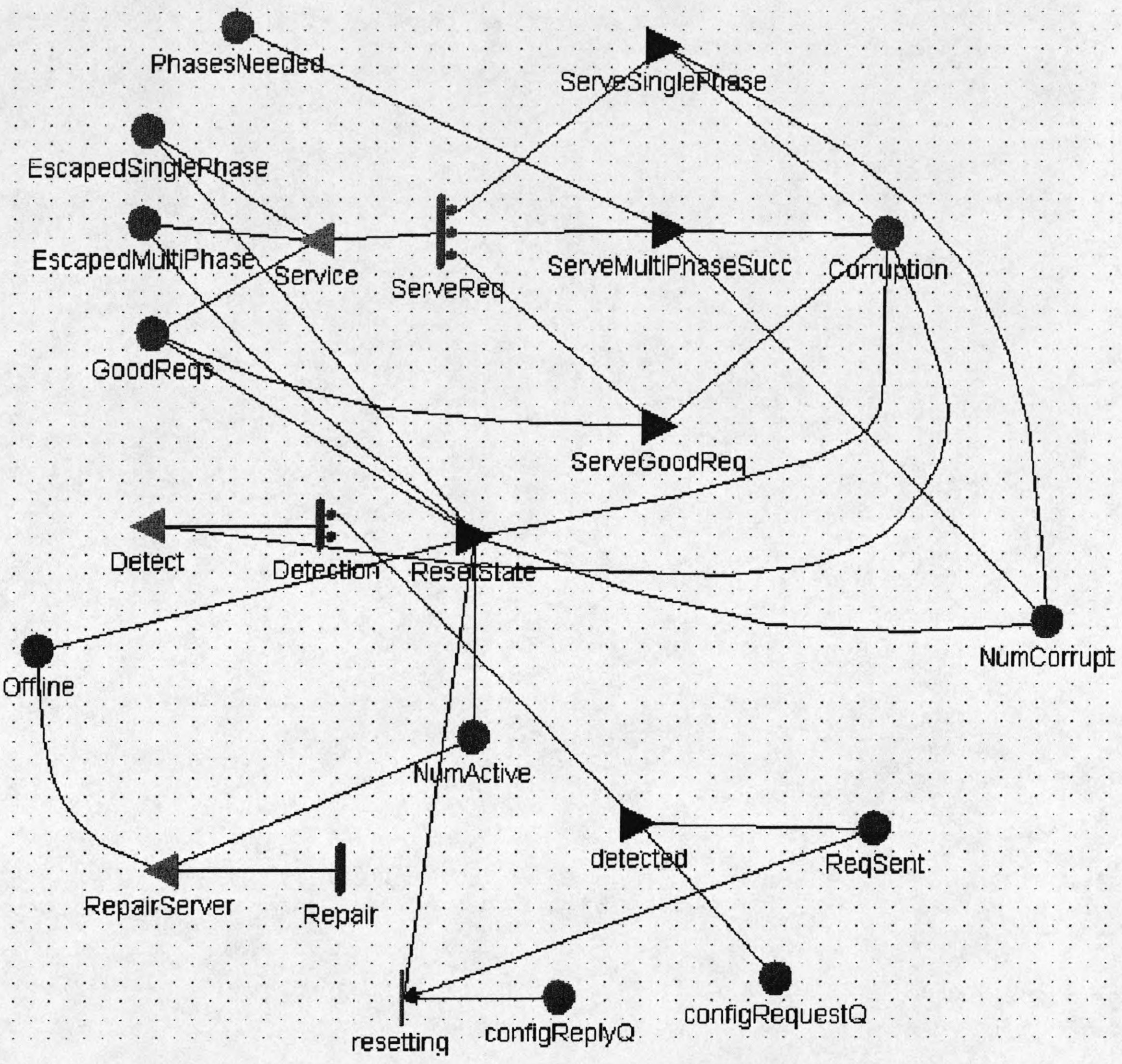
Input Gate:	**Filter**
Predicate	`(EscapedSinglePhase->Mark() +  GoodReqs->Mark()) < MaxGwQLen && Offline->Mark()==0`
Function	`;`

Output Gate:	**MPAttack**

Function

```
short x = MaxPhases>2?((short)(rand()%(MaxPhases-2))+2):2;
//removed to reduce state space
//short x=1;    //comment this when enabling the previous
line
if(EscapedMultiPhase->Mark()==0) {
        EscapedMultiPhase->Mark() = x;
}
else {
        // prob of choosing num phases inversely prop to #
phases
        short y = x + EscapedMultiPhase->Mark();
        short z = 1 + (rand()%(y-1));                  //removed
to reduce state space
        //short z=1;  //comment this when enabling the
previous line
        if(z > x) {
                EscapedMultiPhase->Mark() = x;
        }
}
PhasesNeeded->Mark() = 1+(short)(EscapedMultiPhase->Mark() *
fracPhases);
```

Model: Server



Place Names	Initial Markings
Corruption	0
EscapedMultiPhase	0
EscapedSinglePhase	0
GoodReqs	0
NumActive	NumServers
NumCorrupt	0
Offline	0
PhasesNeeded	0
ReqSent	0
configReplyQ	0
configRequestQ	0

Timed Activity:	Detection
Exponential Distribution Parameters	**Rate** DetectionRate
Activation Predicate	1
Reactivation Predicate	1

Case Distributions	**case 1**
	(ProbDetection*Corruption->Mark())/MaxPhases
	case 2
	1-(ProbDetection*Corruption->Mark())/MaxPhases

Timed Activity:	Repair
Exponential Distribution Parameters	**Rate** RepairRate
Activation Predicate	(none)
Reactivation Predicate	(none)

Timed Activity:	ServeReq
Exponential Distribution Parameters	**Rate** ServiceRate
Activation Predicate	1
Reactivation Predicate	1

Case Distributions	**case 1**
	(EscapedSinglePhase->Mark()*1.0)/(EscapedSinglePhase->Mark() + EscapedMultiPhase->Mark() + GoodReqs->Mark())
	case 2
	(EscapedMultiPhase->Mark())/(EscapedSinglePhase->Mark() + EscapedMultiPhase->Mark() + GoodReqs->Mark())
	case 3
	(GoodReqs->Mark()*1.0)/(EscapedSinglePhase->Mark() + EscapedMultiPhase->Mark() + GoodReqs->Mark())

Instantaneous Activities Without Cases:
resetting

Input Gate:	Detect
Predicate	Corruption->Mark() > 0 && Offline->Mark()==0 && ReqSent->Mark()==0
Function	;

Input Gate:	RepairServer
Predicate	Offline->Mark()
Function	

```
Offline->Mark()=0;
NumActive->Mark()++;
```

Input Gate: **Service**

Predicate

```
(EscapedSinglePhase->Mark() + EscapedMultiPhase->Mark() +
GoodReqs->Mark()) > 0
```

Function

```
;
```

Output Gate: **ResetState**

Function

```
Offline->Mark()=1; // take offline right now instead of
waiting for CM reply
EscapedSinglePhase->Mark()=0;
EscapedMultiPhase->Mark()=0;
GoodReqs->Mark()=0;
NumActive->Mark()--;
if(Corruption->Mark()==MaxPhases) NumCorrupt->Mark()--;
Corruption->Mark()=0;
```

Output Gate: **ServeGoodReq**

Function

```
if(GoodReqs->Mark()) {
        GoodReqs->Mark()--;
}
```

Output Gate: **ServeMultiPhaseSucc**

Function

```
if(EscapedMultiPhase->Mark()) {
        EscapedMultiPhase->Mark()--;
        short flag = 0;
        if(Corruption->Mark() < PhasesNeeded->Mark()) {
                Corruption->Mark()++;
                flag=1;
        }
        if(Corruption->Mark() >= PhasesNeeded->Mark())
{
                EscapedMultiPhase->Mark()=0;
                PhasesNeeded->Mark() = 0;
                Corruption->Mark() = MaxPhases;
                if (flag==1)
                        NumCorrupt->Mark()++;
        }
}
```

Output Gate: **ServeSinglePhase**

Function

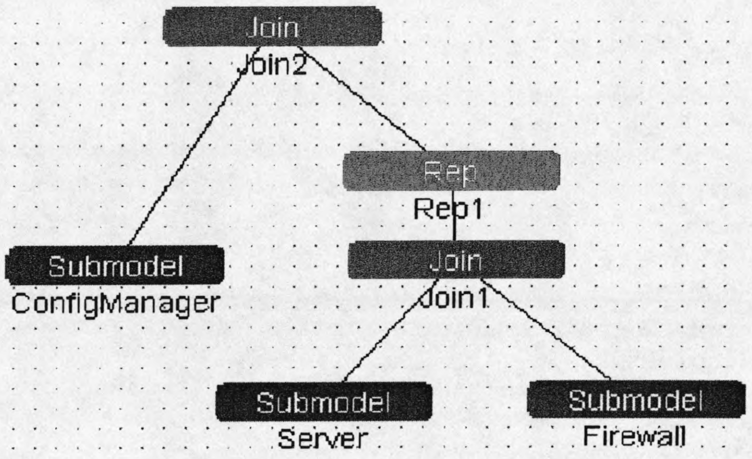
```
if(EscapedSinglePhase->Mark()) {
        EscapedSinglePhase->Mark()--;
        if (Corruption->Mark() < MaxPhases)
                NumCorrupt->Mark()++;
        Corruption->Mark() = MaxPhases;
}
```

Output Gate: **detected**

Function

```
configRequestQ->Mark() = 1;
ReqSent->Mark() = 1;
```

Model: MRCM



Rep Node	Reps	Shared State Variables
Rep1	NumServers	NumActive
		NumCorrupt
		configReplyQ
		configRequestQ

Join Node: Join1 :

State Variable Name	Submodel Variables
EscapedMultiPhase	Server->EscapedMultiPhase
	Firewall->EscapedMultiPhase
EscapedSinglePhase	Server->EscapedSinglePhase
	Firewall->EscapedSinglePhase
GoodReqs	Server->GoodReqs
	Firewall->GoodReqs
NumActive	Server->NumActive
	Firewall->NumActive
NumCorrupt	Server->NumCorrupt
Offline	Server->Offline
	Firewall->Offline
PhasesNeeded	Server->PhasesNeeded
	Firewall->PhasesNeeded
configReplyQ	Server->configReplyQ
configRequestQ	Server->configRequestQ

Join Node: Join2 :

State Variable Name	Submodel Variables
configReplyQ	Rep1->configReplyQ
	ConfigManager->configReplyQ
configRequestQ	Rep1->configRequestQ
	ConfigManager->configRequestQ

Performance Variable Model: MRCM_PV		
Top Level Model Information	Child Model Name	MRCM
	Model Type	Rep/Join

Performance Variable : throughput			
Affecting Models	Server		
Impulse Functions	Server->ServeReq_case3		
	(Reward is over all Available Models) `return 1;`		
Reward Function	*(Reward is over all Available Models)*		
Simulator Statistics	Type	Time Averaged Interval of Time	
	Options	Estimate Mean	
		Include Lower Bound on Interval Estimate	
		Include Upper Bound on Interval Estimate	
		Estimate out of Range Probabilities	
		Confidence Level is Relative	
	Parameters	Start Time	0.0
		Stop Time	30.0
	Confidence	Confidence Level	0.95
		Confidence Interval	0.01

Performance Variable : arrivalrate			
Affecting Models	Firewall		
Impulse Functions	Firewall->FilterRequests_case1		
	(Reward is over all Available Models) `return 1;`		
	Firewall->FilterRequests_case2		
	(Reward is over all Available Models) `return (1+(MaxPhases/2.0));`		
	Firewall->FilterRequests_case3		
	(Reward is over all Available Models) `return 1;`		
Reward Function	*(Reward is over all Available Models)*		
Simulator Statistics	Type	Time Averaged Interval of Time	
	Options	Estimate Mean	
		Include Lower Bound on Interval Estimate	
		Include Upper Bound on Interval Estimate	
		Estimate out of Range Probabilities	
		Confidence Level is Relative	
	Parameters	Start Time	0.0
		Stop Time	30.0
	Confidence	Confidence Level	0.95
		Confidence Interval	0.01

Performance Variable : fracCorruptServers			
Affecting Models	Server		
Impulse Functions			

Reward Function	*(Reward is over all Available Models)*
	`if (Server->NumActive->Mark() > 0)`
	`        return Server->NumCorrupt->Mark()*1.0/(Server->NumActive->Mark()*NumServers);`		

Simulator Statistics	Type	Instant of Time	
	Options	Estimate Mean	
		Include Lower Bound on Interval Estimate	
		Include Upper Bound on Interval Estimate	
		Estimate out of Range Probabilities	
		Confidence Level is Relative	
	Parameters	Start Time	30.0
	Confidence	Confidence Level	0.95
		Confidence Interval	0.01

Performance Variable : StrongUnavailability			
Affecting Models	Server		
Impulse Functions			

Reward Function	*(Reward is over all Available Models)*
	`if(Server->NumCorrupt->Mark()>0 || Server->NumActive->Mark()==0)`
	`        return 1.0/NumServers;`		

Simulator Statistics	Type	Time Averaged Interval of Time	
	Options	Estimate Mean	
		Include Lower Bound on Interval Estimate	
		Include Upper Bound on Interval Estimate	
		Estimate out of Range Probabilities	
		Confidence Level is Relative	
	Parameters	Start Time	0.0
		Stop Time	30.0
	Confidence	Confidence Level	0.95
		Confidence Interval	0.01

Performance Variable : WeakUnavailability		
Affecting Models	Server	
Impulse Functions		

Reward Function	*(Reward is over all Available Models*
	`if (Server->NumActive->Mark() == Server->NumCorrupt->Mark())`
	`        return 1.0/NumServers;`	

Simulator Statistics	Type	Time Averaged Interval of Time
	Options	Estimate Mean
		Include Lower Bound on Interval Estimate
		Include Upper Bound on Interval Estimate
		Estimate out of Range Probabilities

		Confidence Level is Relative	
	Parameters	Start Time	0.0
		Stop Time	30.0
	Confidence	Confidence Level	0.95
		Confidence Interval	0.01

Performance Variable : numCorrupt

Affecting Models	Server		
Impulse Functions			
Reward Function	*(Reward is over all Available Models)* `return (Server->Corruption->Mark()>=MaxPhases?1.0:0);`		
Simulator Statistics	Type	Instant of Time	
	Options	Estimate Mean	
		Include Lower Bound on Interval Estimate	
		Include Upper Bound on Interval Estimate	
		Estimate out of Range Probabilities	
		Confidence Level is Relative	
	Parameters	Start Time	30.0
	Confidence	Confidence Level	0.95
		Confidence Interval	0.01

Performance Variable : goodthroughput

Affecting Models	Server		
Impulse Functions	Server->ServeReq_case3		
	(Reward is over all Available Models) `if (Server->Corruption->Mark() < MaxPhases) return 1;`		
Reward Function	*(Reward is over all Available Models)*		
Simulator Statistics	Type	Time Averaged Interval of Time	
	Options	Estimate Mean	
		Include Lower Bound on Interval Estimate	
		Include Upper Bound on Interval Estimate	
		Estimate out of Range Probabilities	
		Confidence Level is Relative	
	Parameters	Start Time	0.0
		Stop Time	30.0
	Confidence	Confidence Level	0.95
		Confidence Interval	0.01

Performance Variable : badthroughput

Affecting Models	Server	
Impulse Functions	Server->ServeReq_case3	
	(Reward is over all Available Models) `if (Server->Corruption->Mark() == MaxPhases) return 1;`	
Reward Function	*(Reward is over all Available Models)*	
Simulator Statistics	Type	Time Averaged Interval of Time
	Options	Estimate Mean

		Include Lower Bound on Interval Estimate	
		Include Upper Bound on Interval Estimate	
		Estimate out of Range Probabilities	
		Confidence Level is Relative	
	Parameters	Start Time	0.0
		Stop Time	30.0
	Confidence	Confidence Level	0.95
		Confidence Interval	0.01

Performance Variable Model: MRCM_PV_SS		
Top Level Model Information	Child Model Name	MRCM
	Model Type	Rep/Join

Performance Variable : SSfracCorruptServers			
Affecting Models	Server		
Impulse Functions			

Reward Function	*(Reward is over all Available Models)*
	`if (Server->NumActive->Mark() > 0)`
	`return Server->NumCorrupt->Mark()*1.0/(Server->NumActive->Mark()*NumServers);`		

Simulator Statistics	Type	Steady State	
	Options	Estimate Mean	
		Include Lower Bound on Interval Estimate	
		Include Upper Bound on Interval Estimate	
		Estimate out of Range Probabilities	
		Confidence Level is Relative	
	Parameters	Initial Transient	5.0
		Batch Size	1000.0
	Confidence	Confidence Level	0.95
		Confidence Interval	0.01

Performance Variable : SSStrongUnavailability			
Affecting Models	Server		
Impulse Functions			

Reward Function	*(Reward is over all Available Models)*
	`if(Server->NumCorrupt->Mark()>0 || Server->NumActive->Mark()==0)`
	`return 1.0/NumServers;`		

Simulator Statistics	Type	Steady State	
	Options	Estimate Mean	
		Include Lower Bound on Interval Estimate	
		Include Upper Bound on Interval Estimate	
		Estimate out of Range Probabilities	
		Confidence Level is Relative	
	Parameters	Initial Transient	5.0
		Batch Size	1000.0
	Confidence	Confidence Level	0.95
		Confidence Interval	0.01

Performance Variable : SSWeakUnavailability			
Affecting Models	Server		
Impulse Functions			

Reward Function	*(Reward is over all Available Models)*
	`if (Server->NumActive->Mark() == Server->NumCorrupt->Mark())`
	`        return 1.0/NumServers;`		

Simulator Statistics	Type	Steady State	
	Options	Estimate Mean	
		Include Lower Bound on Interval Estimate	
		Include Upper Bound on Interval Estimate	
		Estimate out of Range Probabilities	
		Confidence Level is Relative	
	Parameters	Initial Transient	5.0
		Batch Size	1000.0
	Confidence	Confidence Level	0.95
		Confidence Interval	0.01

Performance Variable : SSnumCorrupt			
Affecting Models	Server		
Impulse Functions			
Reward Function	*(Reward is over all Available Models)* `return (Server->Corruption->Mark()>=MaxPhases?1.0:0);`		
Simulator Statistics	Type	Steady State	
	Options	Estimate Mean	
		Include Lower Bound on Interval Estimate	
		Include Upper Bound on Interval Estimate	
		Estimate out of Range Probabilities	
		Confidence Level is Relative	
	Parameters	Initial Transient	5.0
		Batch Size	1000.0
	Confidence	Confidence Level	0.95
		Confidence Interval	0.01

Performance Variable : SSgoodthroughput		
Affecting Models	Server	
Impulse Functions	Server->ServeReq_case3	

	(Reward is over all Available Models)
	`if (Server->Corruption->Mark() < MaxPhases)`
	`        return 1;`	

Reward Function	*(Reward is over all Available Models)*	
Simulator Statistics	Type	Steady State
	Options	Estimate Mean
		Include Lower Bound on Interval Estimate

		Include Upper Bound on Interval Estimate	
		Estimate out of Range Probabilities	
		Confidence Level is Relative	
	Parameters	Initial Transient	5.0
		Batch Size	1000.0
	Confidence	Confidence Level	0.95
		Confidence Interval	0.01

Performance Variable : SSbadthroughput

Affecting Models	Server		
Impulse Functions	Server->ServeReq_case3		
	(Reward is over all Available Models) `if (Server->Corruption->Mark() == MaxPhases) return 1;`		
Reward Function	*(Reward is over all Available Models)*		
Simulator Statistics	Type	Steady State	
	Options	Estimate Mean	
		Include Lower Bound on Interval Estimate	
		Include Upper Bound on Interval Estimate	
		Estimate out of Range Probabilities	
		Confidence Level is Relative	
	Parameters	Initial Transient	5.0
		Batch Size	1000.0
	Confidence	Confidence Level	0.95
		Confidence Interval	0.01

Range Study Variable Assignments for Study *Detection* in Project *MRCM* :

Variable	Type	Range Type	Range	Increment	Increment Type	Function	n
ConfMgrRate	double	Fixed	10	-	-	-	-
DetectionRate	double	Fixed	1	-	-	-	-
FilterRate	double	Fixed	10	-	-	-	-
MaxGwQLen	short	Fixed	10	-	-	-	-
MaxPhases	short	Fixed	5	-	-	-	-
NumServers	short	Fixed	11	-	-	-	-
ProbDetection	double	Manual	[0, .1, .2, .3, .4, .6, .8, 1]	-	-	-	-
ProbFiltering	double	Fixed	.4	-	-	-	-
ProbMultiPhase	double	Fixed	.001	-	-	-	-
ProbSinglePhase	double	Fixed	.002	-	-	-	-
RepairRate	double	Fixed	.06	-	-	-	-
ServiceRate	double	Fixed	16	-	-	-	-
fracPhases	double	Fixed	.6	-	-	-	-

Range Study Variable Assignments for Study *MRCM_Study* in Project *MRCM* :

Variable	Type	Range Type	Range	Increment	Increment Type	Function	n
ConfMgrRate	double	Fixed	10	-	-	-	-
DetectionRate	double	Fixed	1	-	-	-	-
FilterRate	double	Fixed	10	-	-	-	-
MaxGwQLen	short	Fixed	10	-	-	-	-
MaxPhases	short	Fixed	5	-	-	-	-
NumServers	short	Fixed	11	-	-	-	-
ProbDetection	double	Fixed	.5	-	-	-	-
ProbFiltering	double	Fixed	.4	-	-	-	-
ProbMultiPhase	double	Fixed	.001	-	-	-	-
ProbSinglePhase	double	Fixed	.002	-	-	-	-
RepairRate	double	Fixed	.06	-	-	-	-
ServiceRate	double	Fixed	16	-	-	-	-
fracPhases	double	Fixed	.6	-	-	-	-

Range Study Variable Assignments for Study *MRCM_Study_SS* in Project *MRCM* :

Variable	Type	Range Type	Range	Increment	Increment Type	Function	n
ConfMgrRate	double	Fixed	10	-	-	-	-
DetectionRate	double	Fixed	1	-	-	-	-
FilterRate	double	Fixed	10	-	-	-	-
MaxGwQLen	short	Fixed	10	-	-	-	-
MaxPhases	short	Fixed	5	-	-	-	-
NumServers	short	Fixed	11	-	-	-	-
ProbDetection	double	Fixed	.5	-	-	-	-
ProbFiltering	double	Fixed	.4	-	-	-	-
ProbMultiPhase	double	Fixed	.001	-	-	-	-
ProbSinglePhase	double	Fixed	.002	-	-	-	-
RepairRate	double	Fixed	.06	-	-	-	-
ServiceRate	double	Fixed	16	-	-	-	-
fracPhases	double	Fixed	.6	-	-	-	-

Range Study Variable Assignments for Study *MultiPhase* in Project *MRCM* :

Variable	Type	Range Type	Range	Increment	Increment Type	Function	n
ConfMgrRate	double	Fixed	10	-	-	-	-
DetectionRate	double	Fixed	1	-	-	-	-
FilterRate	double	Fixed	10	-	-	-	-
MaxGwQLen	short	Fixed	10	-	-	-	-
MaxPhases	short	Fixed	5	-	-	-	-
NumServers	short	Fixed	11	-	-	-	-
ProbDetection	double	Fixed	.5	-	-	-	-
ProbFiltering	double	Fixed	.4	-	-	-	-
ProbMultiPhase	double	Incremental	[0.0,0.009000000000000001]	.001	Additive	-	-
ProbSinglePhase	double	Fixed	0	-	-	-	-
RepairRate	double	Fixed	.06	-	-	-	-
ServiceRate	double	Fixed	16	-	-	-	-
fracPhases	double	Fixed	.6	-	-	-	-

Range Study Variable Assignments for Study *NumServers* in Project *MRCM* :

Variable	Type	Range Type	Range	Increment	Increment Type	Function	n
ConfMgrRate	double	Fixed	10	-	-	-	-
DetectionRate	double	Fixed	1	-	-	-	-
FilterRate	double	Fixed	10	-	-	-	-
MaxGwQLen	short	Fixed	10	-	-	-	-
MaxPhases	short	Fixed	5	-	-	-	-
NumServers	short	Incremental	[3,12]	3	Additive	-	-
ProbDetection	double	Fixed	.5	-	-	-	-
ProbFiltering	double	Fixed	.4	-	-	-	-
ProbMultiPhase	double	Fixed	.001	-	-	-	-
ProbSinglePhase	double	Fixed	.002	-	-	-	-
RepairRate	double	Fixed	.06	-	-	-	-
ServiceRate	double	Fixed	16	-	-	-	-
fracPhases	double	Fixed	.6	-	-	-	-

Range Study Variable Assignments for Study *Repair* in Project *MRCM*:

Variable	Type	Range Type	Range	Increment	Increment Type	Function	n
ConfMgrRate	double	Fixed	10	-	-	-	-
DetectionRate	double	Fixed	1	-	-	-	-
FilterRate	double	Fixed	10	-	-	-	-
MaxGwQLen	short	Fixed	10	-	-	-	-
MaxPhases	short	Fixed	5	-	-	-	-
NumServers	short	Fixed	11	-	-	-	-
ProbDetection	double	Fixed	.5	-	-	-	-
ProbFiltering	double	Fixed	.4	-	-	-	-
ProbMultiPhase	double	Fixed	.001	-	-	-	-
ProbSinglePhase	double	Fixed	.002	-	-	-	-
RepairRate	double	Manual	[0, .01, .02, .03, .04, .05, .1, .15, .2]	-	-	-	-
ServiceRate	double	Fixed	16	-	-	-	-
fracPhases	double	Fixed	.6	-	-	-	-

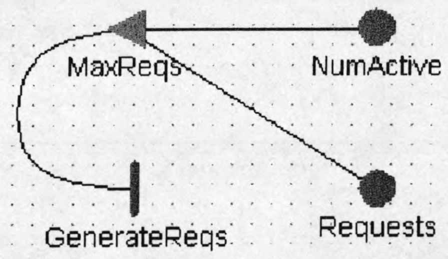
Range Study Variable Assignments for Study *SinglePhase* in Project *MRCM*:

Variable	Type	Range Type	Range	Increment	Increment Type	Function	n
ConfMgrRate	double	Fixed	10	-	-	-	-
DetectionRate	double	Fixed	1	-	-	-	-
FilterRate	double	Fixed	10	-	-	-	-
MaxGwQLen	short	Fixed	10	-	-	-	-
MaxPhases	short	Fixed	5	-	-	-	-
NumServers	short	Fixed	11	-	-	-	-
ProbDetection	double	Fixed	.5	-	-	-	-
ProbFiltering	double	Fixed	.4	-	-	-	-
ProbMultiPhase	double	Fixed	0	-	-	-	-
ProbSinglePhase	double	Incremental	[0.0,0.009000000000000001]	0.001	Additive	-	-
RepairRate	double	Fixed	.06	-	-	-	-
ServiceRate	double	Fixed	16	-	-	-	-
fracPhases	double	Fixed	.6	-	-	-	-

APPENDIX A.3

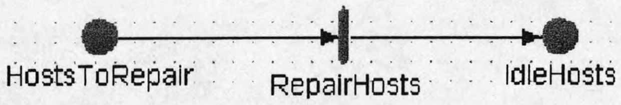
SAN Modeling of the State Machine Replication (SMR) Architecture

Model: Client



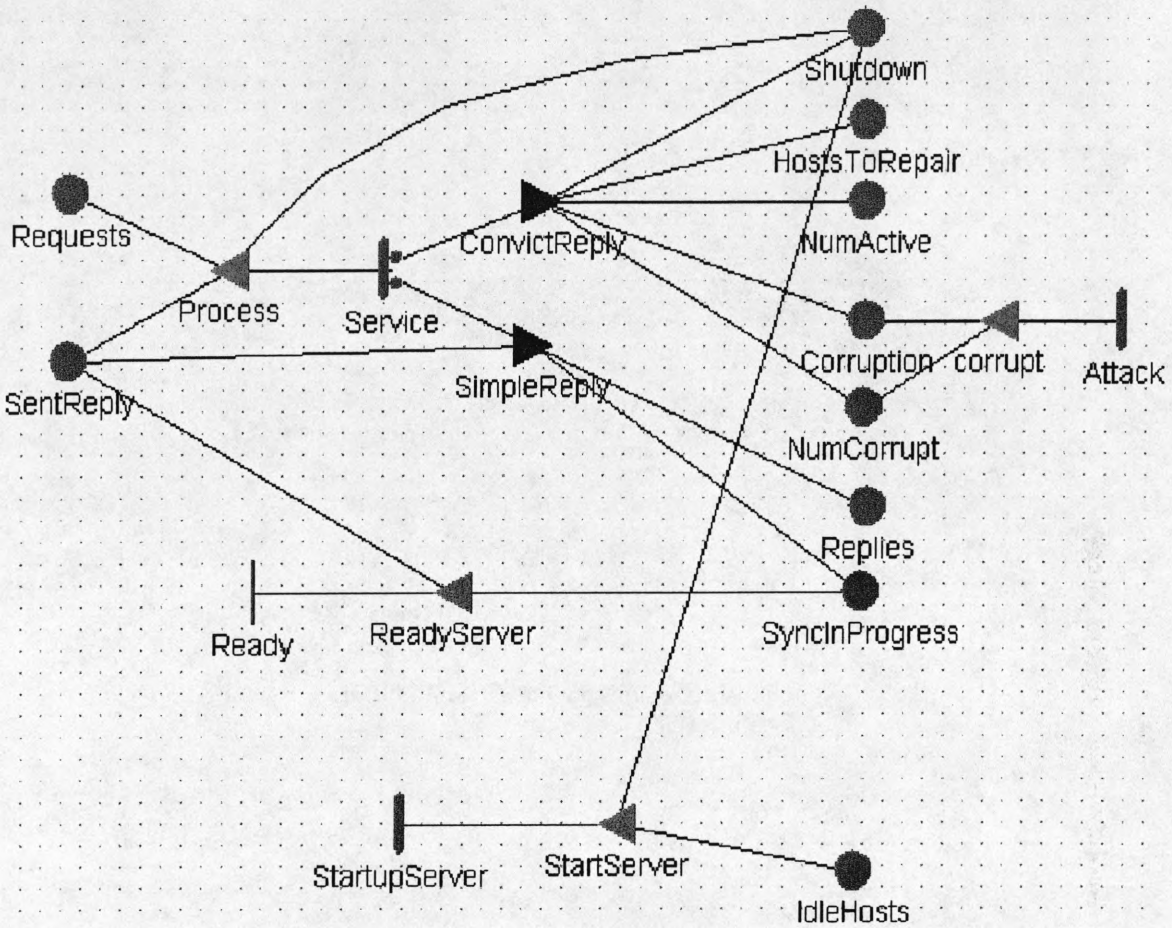
Place Names	Initial Markings
NumActive	NumReps
Requests	0
Timed Activity:	**GenerateReqs**
Exponential Distribution Parameters	**Rate** `GenReqRate`
Activation Predicate	(none)
Reactivation Predicate	(none)
Input Gate:	**MaxReqs**
Predicate	`(Requests->Mark()) ==0`
Function	`Requests->Mark() = NumActive->Mark();`

Model: ManualRepair



Place Names	Initial Markings
HostsToRepair	0
IdleHosts	NumHosts-NumReps
Timed Activity:	**RepairHosts**
Exponential Distribution Parameters	**Rate** `HostRepairRate`
Activation Predicate	(none)
Reactivation Predicate	(none)

Model: Server



Place Names	Initial Markings
Corruption	0
HostsToRepair	0
IdleHosts	NumHosts-NumReps
NumActive	NumReps
NumCorrupt	0
Replies	0
Requests	0
SentReply	0
Shutdown	0
SyncInProgress	0
Timed Activity:	Attack
Exponential Distribution Parameters	**Rate** `AttackRate`
Activation Predicate	(none)
Reactivation Predicate	(none)

Timed Activity:	Service
Exponential Distribution Parameters	**Rate** `ServiceRate`
Activation Predicate	`1`
Reactivation Predicate	`1`

Case Distributions	**case 1**
	`(Corruption->Mark()>0)?((3*NumCorrupt->Mark() < NumActive->Mark())?ProbMisbehavior:0):0`
	case 2
	`(Corruption->Mark()>0)?((3*NumCorrupt->Mark() < NumActive->Mark())?(1-ProbMisbehavior):1):1`

Timed Activity:	StartupServer
Exponential Distribution Parameters	**Rate** `ServerStartRate`
Activation Predicate	(none)
Reactivation Predicate	(none)

Instantaneous Activities Without Cases:

Ready

Input Gate:	Process
Predicate	`Requests->Mark()>0 && SentReply->Mark()==0 && Shutdown->Mark()==0`

Function	`Requests->Mark()--;`
	`SentReply->Mark()=1;`

Input Gate:	ReadyServer
Predicate	`SyncInProgress->Mark()==0 && SentReply->Mark()==1`
Function	`SentReply->Mark()=0;`

Input Gate:	StartServer
Predicate	`Shutdown->Mark() && IdleHosts->Mark()>0`

Function	`Shutdown->Mark()=0;`
	`IdleHosts->Mark()--;`
	`SentReply->Mark()=0;`
	`NumActive->Mark()++;`
	`Corruption->Mark()=0;`

Input Gate:	corrupt
Predicate	`Corruption->Mark()==0`
Function	

	`Corruption->Mark()=1;`
	`NumCorrupt->Mark()++;`

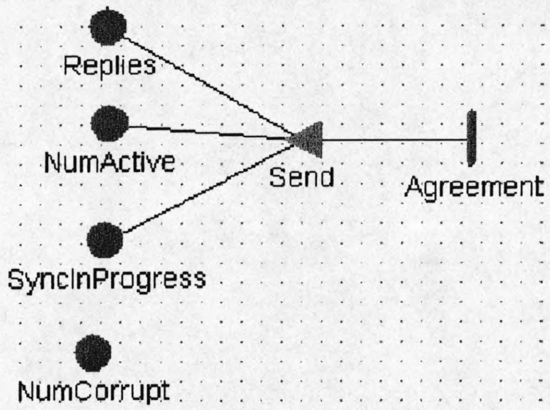
Output Gate:	ConvictReply

Function	`Shutdown->Mark()=1;`
	`NumActive->Mark()--;`
	`NumCorrupt->Mark()--;`
	`HostsToRepair->Mark()++;`

Output Gate:	SimpleReply

Function	`Replies->Mark()++;`
	`if(SyncInProgress->Mark()==0) SyncInProgress->Mark()=1;`
	`SentReply->Mark()=1;`

Model: Synchronizer



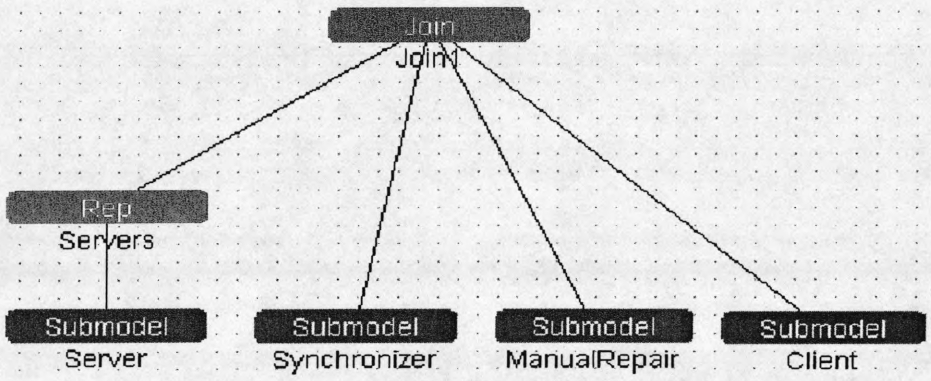
Place Names	**Initial Markings**
NumActive	NumReps
NumCorrupt	0
Replies	0
SyncInProgress	0

Timed Activity:	Agreement
Exponential Distribution Parameters	**Rate** `1.0/(AgreementDelay+IncrementalDelay*((NumActive->Mark()-1)/3))`
Activation Predicate	1
Reactivation Predicate	1

Input Gate:	**Send**
Predicate	`Replies->Mark()==NumActive->Mark()`

Function	`SyncInProgress->Mark()=0;`
	`Replies->Mark()=0;`

Model: SMR



Rep Node	Reps	Shared State Variables
Servers	NumReps	HostsToRepair
		IdleHosts
		NumActive
		NumCorrupt
		Replies
		Requests
		SyncInProgress

Join Node: Join1 :

State Variable Name	Submodel Variables
HostsToRepair	Servers->HostsToRepair
	ManualRepair->HostsToRepair
IdleHosts	Servers->IdleHosts
	ManualRepair->IdleHosts
NumActive	Servers->NumActive
	Synchronizer->NumActive
	Client->NumActive
NumCorrupt	Servers->NumCorrupt
	Synchronizer->NumCorrupt
Replies	Servers->Replies
	Synchronizer->Replies
Requests	Servers->Requests
	Client->Requests
SyncInProgress	Servers->SyncInProgress
	Synchronizer->SyncInProgress

Performance Variable Model: SMR_PV		
Top Level Model Information	Child Model Name	SMR
	Model Type	Rep/Join

Performance Variable : arrivalrate			
Affecting Models	Client		
Impulse Functions	Client->GenerateReqs		
	(Reward is over all Available Models)		
	`return 1;`		
Reward Function	*(Reward is over all Available Models)*		
Simulator Statistics	Type	Time Averaged Interval of Time	
	Options	Estimate Mean	
		Include Lower Bound on Interval Estimate	
		Include Upper Bound on Interval Estimate	
		Estimate out of Range Probabilities	
		Confidence Level is Relative	
	Parameters	Start Time	0.0
		Stop Time	30.0
	Confidence	Confidence Level	0.95
		Confidence Interval	0.01

Performance Variable : totalthroughput			
Affecting Models	Synchronizer		
Impulse Functions	Synchronizer->Agreement		
	(Reward is over all Available Models)		
	`return 1;`		
Reward Function	*(Reward is over all Available Models)*		
Simulator Statistics	Type	Time Averaged Interval of Time	
	Options	Estimate Mean	
		Include Lower Bound on Interval Estimate	
		Include Upper Bound on Interval Estimate	
		Estimate out of Range Probabilities	
		Confidence Level is Relative	
	Parameters	Start Time	0.0
		Stop Time	30.0
	Confidence	Confidence Level	0.95
		Confidence Interval	0.01

Performance Variable : Unavailability			
Affecting Models	Server		
Impulse Functions			

Reward Function	*(Reward is over all Available Models)*
	`if((3*Server->NumCorrupt->Mark()+1)>Server->NumActive->Mark())`
	`        return 1.0/NumReps;`		

Simulator Statistics	Type	Time Averaged Interval of Time	
	Options	Estimate Mean	
		Include Lower Bound on Interval Estimate	
		Include Upper Bound on Interval Estimate	
		Estimate out of Range Probabilities	
		Confidence Level is Relative	
	Parameters	Start Time	0.0
		Stop Time	30.0
	Confidence	Confidence Level	0.95
		Confidence Interval	0.01

Performance Variable : goodthroughput			
Affecting Models	Synchronizer		
Impulse Functions	Synchronizer->Agreement		

	(Reward is over all Available Models)
	`if (3*Synchronizer->NumCorrupt->Mark() < Synchronizer->NumActive->Mark())`
	`        return 1;`		

Reward Function	*(Reward is over all Available Models)*		
Simulator Statistics	Type	Time Averaged Interval of Time	
	Options	Estimate Mean	
		Include Lower Bound on Interval Estimate	
		Include Upper Bound on Interval Estimate	
		Estimate out of Range Probabilities	
		Confidence Level is Relative	
	Parameters	Start Time	0.0
		Stop Time	30.0
	Confidence	Confidence Level	0.95
		Confidence Interval	0.01

Performance Variable : badthroughput	
Affecting Models	Synchronizer
Impulse Functions	Synchronizer->Agreement

	(Reward is over all Available Models)
	`if (3*Synchronizer->NumCorrupt->Mark() >= Synchronizer->NumActive->Mark())`
	`        return 1;`

Reward Function	*(Reward is over all Available Models)*

Simulator Statistics	Type	Time Averaged Interval of Time	
	Options	Estimate Mean	
		Include Lower Bound on Interval Estimate	
		Include Upper Bound on Interval Estimate	
		Estimate out of Range Probabilities	
		Confidence Level is Relative	
	Parameters	Start Time	0.0
		Stop Time	30.0
	Confidence	Confidence Level	0.95
		Confidence Interval	0.01

Performance Variable Model: SMR_PV_SS

Top Level Model Information	Child Model Name	SMR
	Model Type	Rep/Join

Performance Variable : SSUnavailability

Affecting Models	Server		
Impulse Functions			
Reward Function	*(Reward is over all Available Models)* `if((3*Server->NumCorrupt->Mark()+1)>Server->NumActive->Mark()) return 1.0/NumReps;`		
Simulator Statistics	Type	Steady State	
	Options	Estimate Mean	
		Include Lower Bound on Interval Estimate	
		Include Upper Bound on Interval Estimate	
		Estimate out of Range Probabilities	
		Confidence Level is Relative	
	Parameters	Initial Transient	5.0
		Batch Size	1000.0
	Confidence	Confidence Level	0.95
		Confidence Interval	0.01

Performance Variable : SSgoodthroughput

Affecting Models	Synchronizer
Impulse Functions	Synchronizer->Agreement
	(Reward is over all Available Models) `if (3*Synchronizer->NumCorrupt->Mark() < Synchronizer->NumActive->Mark()) return 1;`
Reward Function	*(Reward is over all Available Models)*

Simulator Statistics	Type	Steady State	
	Options	Estimate Mean	
		Include Lower Bound on Interval Estimate	
		Include Upper Bound on Interval Estimate	
		Estimate out of Range Probabilities	
		Confidence Level is Relative	
	Parameters	Initial Transient	5.0
		Batch Size	1000.0
	Confidence	Confidence Level	0.95
		Confidence Interval	0.01

Performance Variable : SSbadthroughput

Affecting Models	Synchronizer
Impulse Functions	Synchronizer->Agreement
	(Reward is over all Available Models)
	`if (3*Synchronizer->NumCorrupt->Mark() >= Synchronizer->NumActive->Mark())` `return 1;`
Reward Function	*(Reward is over all Available Models)*

Simulator Statistics	Type	Steady State	
	Options	Estimate Mean	
		Include Lower Bound on Interval Estimate	
		Include Upper Bound on Interval Estimate	
		Estimate out of Range Probabilities	
		Confidence Level is Relative	
	Parameters	Initial Transient	5.0
		Batch Size	1000.0
	Confidence	Confidence Level	0.95
		Confidence Interval	0.01

Range Study Variable Assignments for Study *Attack* in Project *SMR-project* :

Variable	Type	Range Type	Range	Increment	Increment Type	Function	n
AgreementDelay	double	Fixed	.01	-	-	-	-
AttackRate	double	Incremental	[0.0,0.09999999999999999]	.01	Additive	-	-
GenReqRate	double	Fixed	100	-	-	-	-
HostRepairRate	double	Fixed	0.02	-	-	-	-
IncrementalDelay	double	Fixed	.005	-	-	-	-
NumHosts	short	Fixed	15	-	-	-	-
NumReps	short	Fixed	12	-	-	-	-
ProbMisbehavior	double	Fixed	.5	-	-	-	-
ServerStartRate	double	Fixed	2	-	-	-	-
ServiceRate	double	Fixed	19	-	-	-	-

Range Study Variable Assignments for Study ***SMR_Study*** in Project *SMR-project* :

Variable	Type	Range Type	Range	Increment	Increment Type	Function	n
AgreementDelay	double	Fixed	.01	-	-	-	-
AttackRate	double	Fixed	.03	-	-	-	-
GenReqRate	double	Fixed	100	-	-	-	-
HostRepairRate	double	Fixed	0.02	-	-	-	-
IncrementalDelay	double	Fixed	.005	-	-	-	-
NumHosts	short	Fixed	15	-	-	-	-
NumReps	short	Fixed	12	-	-	-	-
ProbMisbehavior	double	Fixed	.5	-	-	-	-
ServerStartRate	double	Fixed	2	-	-	-	-
ServiceRate	double	Fixed	19	-	-	-	-

Range Study Variable Assignments for Study ***SMR_Study_SS*** in Project *SMR-project* :

Variable	Type	Range Type	Range	Increment	Increment Type	Function	n
AgreementDelay	double	Fixed	.01	-	-	-	-
AttackRate	double	Fixed	.03	-	-	-	-
GenReqRate	double	Fixed	100	-	-	-	-
HostRepairRate	double	Fixed	.02	-	-	-	-
IncrementalDelay	double	Fixed	.005	-	-	-	-
NumHosts	short	Fixed	15	-	-	-	-
NumReps	short	Fixed	12	-	-	-	-
ProbMisbehavior	double	Fixed	.5	-	-	-	-
ServerStartRate	double	Fixed	2	-	-	-	-
ServiceRate	double	Fixed	12	-	-	-	-

Range Study Variable Assignments for Study ***Misbehavior*** in Project *SMR-project* :

Variable	Type	Range Type	Range	Increment	Increment Type	Function	n
AgreementDelay	double	Fixed	.01	-	-	-	-
AttackRate	double	Fixed	.03	-	-	-	-
GenReqRate	double	Fixed	100	-	-	-	-
HostRepairRate	double	Fixed	0.02	-	-	-	-
IncrementalDelay	double	Fixed	.005	-	-	-	-
NumHosts	short	Fixed	15	-	-	-	-
NumReps	short	Fixed	12	-	-	-	-
ProbMisbehavior	double	Manual	[0, .1, .2, .3, .4, .6, .8, 1]	-	-	-	-
ServerStartRate	double	Fixed	2	-	-	-	-
ServiceRate	double	Fixed	19	-	-	-	-

Range Study Variable Assignments for Study *NumServers* in Project *SMR-project* :

Variable	Type	Range Type	Range	Increment	Increment Type	Function	n
AgreementDelay	double	Fixed	.01	-	-	-	-
AttackRate	double	Fixed	.03	-	-	-	-
GenReqRate	double	Fixed	100	-	-	-	-
HostRepairRate	double	Fixed	0.02	-	-	-	-
IncrementalDelay	double	Fixed	.005	-	-	-	-
NumHosts	short	Incremental	[7,16]	3	Additive	-	-
NumReps	short	Incremental	[4,13]	3	Additive	-	-
ProbMisbehavior	double	Fixed	.5	-	-	-	-
ServerStartRate	double	Fixed	2	-	-	-	-
ServiceRate	double	Fixed	19	-	-	-	-

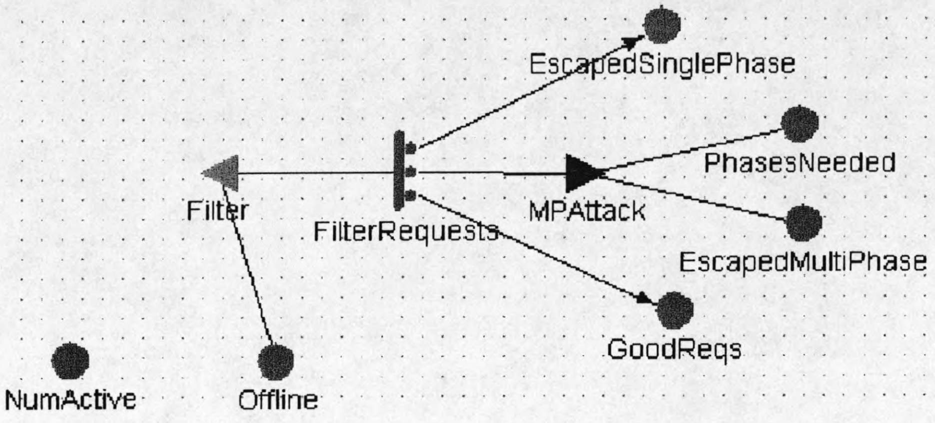
Range Study Variable Assignments for Study *Repair* in Project *SMR-project* :

Variable	Type	Range Type	Range	Increment	Increment Type	Function	n
AgreementDelay	double	Fixed	.01	-	-	-	-
AttackRate	double	Fixed	.03	-	-	-	-
GenReqRate	double	Fixed	100	-	-	-	-
HostRepairRate	double	Manual	[0, .01, .02, .03, .04, .05, .1, .15]	-	-	-	-
IncrementalDelay	double	Fixed	.005	-	-	-	-
NumHosts	short	Fixed	15	-	-	-	-
NumReps	short	Fixed	12	-	-	-	-
ProbMisbehavior	double	Fixed	.5	-	-	-	-
ServerStartRate	double	Fixed	2	-	-	-	-
ServiceRate	double	Fixed	19	-	-	-	-

APPENDIX A.4

SAN Modeling of the Multicast Routing Decentralized Management (MRDM) Architecture

Model: Firewall



Place Names	Initial Markings
EscapedMultiPhase	0
EscapedSinglePhase	0
GoodReqs	0
NumActive	NumServers
Offline	0
PhasesNeeded	0

Timed Activity:	**FilterRequests**
Exponential Distribution Parameters	**Rate** `FilterRate*(NumActive->Mark()?(NumServers*1.0/NumActive->Mark()):0)`
Activation Predicate	`1`
Reactivation Predicate	`1`

Case Distributions	**case 1**
	`ProbFiltering*ProbSinglePhase`
	case 2
	`ProbFiltering * ProbMultiPhase`
	case 3
	`1 - (ProbSinglePhase + ProbMultiPhase)`

Input Gate:	**Filter**

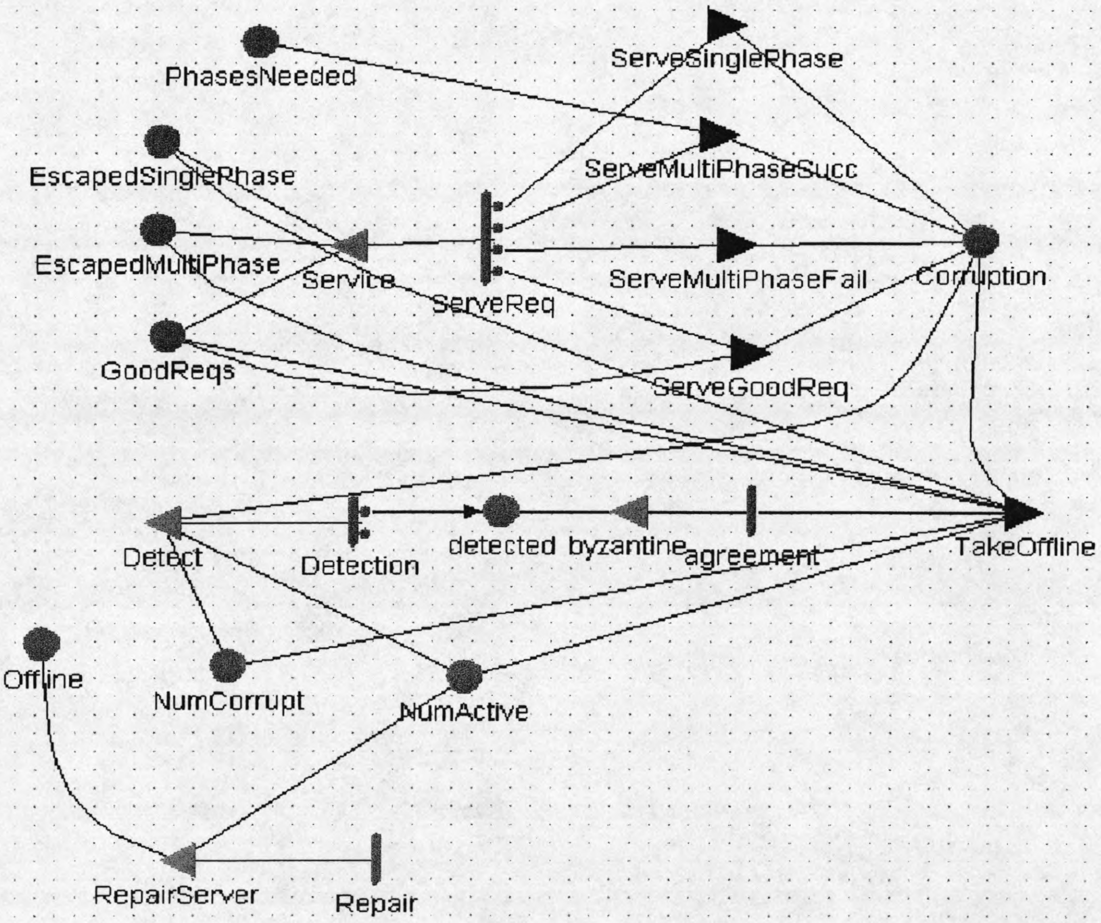
Predicate	`(EscapedSinglePhase->Mark() +  GoodReqs->Mark()) < MaxGwQLen`
	`&& Offline->Mark()==0`

Function	`;`

Output Gate:	MPAttack

Function	
	```
	short x = MaxPhases>2?((short)(rand()%(MaxPhases-2))+2):2;
	//uncomment this
	//short x=1; //comment this
	if(EscapedMultiPhase->Mark()==0) {
	        EscapedMultiPhase->Mark() = x;
	}
	else {
	        // prob of choosing num phases inversely prop to #
	phases
	        short y = x + EscapedMultiPhase->Mark();
	        short z = 1 + (rand()%(y-1)); //uncomment this
	        //short z=1; // comment this
	        if(z > x) {
	                EscapedMultiPhase->Mark() = x;
	        }
	}
	PhasesNeeded->Mark() = 1+(short)(EscapedMultiPhase->Mark()
	* fracPhases);
	```

Model: Server



Place Names	Initial Markings
Corruption	0
EscapedMultiPhase	0
EscapedSinglePhase	0
GoodReqs	0
NumActive	NumServers
NumCorrupt	0
Offline	0
PhasesNeeded	0
detected	0

Timed Activity:	Detection
Exponential Distribution Parameters	**Rate** `DetectionRate`
Activation Predicate	(none)
Reactivation Predicate	(none)
Case Distributions	**case 1** `(ProbDetection*Corruption->Mark())/MaxPhases` **case 2** `1-((ProbDetection*Corruption->Mark())/MaxPhases)`

Timed Activity:	Repair
Exponential Distribution Parameters	**Rate** `RepairRate`
Activation Predicate	(none)
Reactivation Predicate	(none)

Timed Activity:	ServeReq
Exponential Distribution Parameters	**Rate** `ServiceRate`
Activation Predicate	`1`
Reactivation Predicate	`1`
Case Distributions	**case 1** `(EscapedSinglePhase->Mark()*1.0)/(EscapedSinglePhase->Mark() + EscapedMultiPhase->Mark() + GoodReqs->Mark())` **case 2** `(EscapedMultiPhase->Mark()*(NumActive->Mark()?(1.0/NumActive->Mark()):0))/(EscapedSinglePhase->Mark() + EscapedMultiPhase->Mark() + GoodReqs->Mark())` **case 3** `(EscapedMultiPhase->Mark()*(1-(NumActive->Mark()?(1.0/NumActive->Mark()):0)))/(EscapedSinglePhase->Mark() + EscapedMultiPhase->Mark() + GoodReqs->Mark())` **case 4** `(GoodReqs->Mark()*1.0)/(EscapedSinglePhase->Mark() + EscapedMultiPhase->Mark() + GoodReqs->Mark())`

Timed Activity:	**agreement**
Exponential Distribution Parameters	**Rate** `1.0/(AgreementDelay+IncrementalDelay*((NumActive->Mark()-1)/3))`
Activation Predicate	`1`
Reactivation Predicate	`1`
Input Gate:	**Detect**
Predicate	`Corruption->Mark() > 0 && Offline->Mark()==0 && detected->Mark()==0`
Function	`;`
Input Gate:	**RepairServer**
Predicate	`Offline->Mark()`

Function	`Offline->Mark()=0;`
	`NumActive->Mark()++;`

Input Gate:	**Service**

Predicate	`(EscapedSinglePhase->Mark() + EscapedMultiPhase->Mark() + GoodReqs->Mark()) > 0`
	`&& Offline->Mark()==0`

Function	`;`
Input Gate:	**byzantine**
Predicate	`(3*NumCorrupt->Mark() < NumActive->Mark()) && detected->Mark() > 0`
Function	`detected->Mark()=0;`
Output Gate:	**ServeGoodReq**

Function	`if(GoodReqs->Mark()) {`
	        `GoodReqs->Mark()--;`
	`}`

Output Gate:	**ServeMultiPhaseFail**

Function	`if(EscapedMultiPhase->Mark()) {`
	        `EscapedMultiPhase->Mark()--;`
	        `if(EscapedMultiPhase->Mark()==0) {`
	                `if (Corruption->Mark() < MaxPhases) {`
	                        `Corruption->Mark()=0;`
	                `}`
	                `PhasesNeeded->Mark() = 0;`
	        `}`
	`}`

Output Gate: **ServeMultiPhaseSucc**

Function

```
if(EscapedMultiPhase->Mark()) {
        EscapedMultiPhase->Mark()--;
        short flag=0;
        if(Corruption->Mark() < PhasesNeeded->Mark()) {
                Corruption->Mark()++;
                flag=1;
        }
        if(Corruption->Mark() == PhasesNeeded->Mark())
{
                EscapedMultiPhase->Mark()=0;
                PhasesNeeded->Mark() = 0;
                Corruption->Mark() = MaxPhases;
                if(flag) NumCorrupt->Mark()++;
        }
}
```

Output Gate: **ServeSinglePhase**

Function

```
if(EscapedSinglePhase->Mark()) {
        EscapedSinglePhase->Mark()--;
        if(Corruption->Mark() < MaxPhases) {
                Corruption->Mark() = MaxPhases;
                NumCorrupt->Mark()++;
        }
}
```

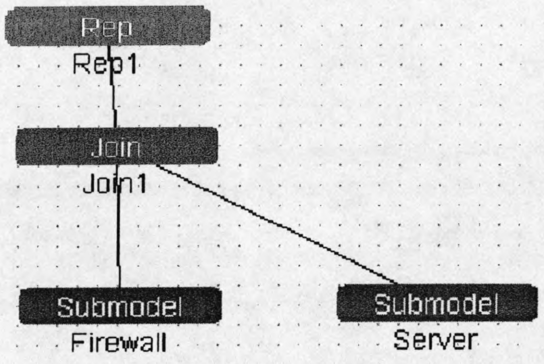
Output Gate: **TakeOffline**

Function

```
EscapedSinglePhase->Mark()=0;
int flag=0;
EscapedMultiPhase->Mark()=0;
GoodReqs->Mark()=0;
Offline->Mark()=1;
NumActive->Mark()--;
if(Corruption->Mark()==MaxPhases) flag=1;
Corruption->Mark()=0;
if (flag ==1) NumCorrupt->Mark()--;

//if(Corruption->Mark()==MaxPhases) NumCorrupt->Mark()-
-;
//Corruption->Mark()=0;
```

Model: MRDM



Rep Node	Reps	Shared State Variables
Rep1	NumServers	NumActive
		NumCorrupt

Join Node: Join1 :

State Variable Name	Submodel Variables
EscapedMultiPhase	Firewall->EscapedMultiPhase
	Server->EscapedMultiPhase
EscapedSinglePhase	Firewall->EscapedSinglePhase
	Server->EscapedSinglePhase
GoodReqs	Firewall->GoodReqs
	Server->GoodReqs
NumActive	Firewall->NumActive
	Server->NumActive
NumCorrupt	Server->NumCorrupt
Offline	Firewall->Offline
	Server->Offline
PhasesNeeded	Firewall->PhasesNeeded
	Server->PhasesNeeded

Performance Variable Model: MRDM_PV

Top Level Model Information	Child Model Name	MRDM
	Model Type	Rep/Join

Performance Variable : throughput

Affecting Models	Server
Impulse Functions	Server->ServeReq_case4 *(Reward is over all Available Models)* `return 1;`
Reward Function	*(Reward is over all Available Models)*

Simulator Statistics	Type	Time Averaged Interval of Time	
	Options	Estimate Mean	
		Include Lower Bound on Interval Estimate	
		Include Upper Bound on Interval Estimate	
		Estimate out of Range Probabilities	
		Confidence Level is Relative	
	Parameters	Start Time	0.0
		Stop Time	30.0
	Confidence	Confidence Level	0.95
		Confidence Interval	0.1

Performance Variable : arrivalrate

Affecting Models	Firewall
Impulse Functions	Firewall->FilterRequests_case1
	(Reward is over all Available Models) `return 1;`
	Firewall->FilterRequests_case2
	(Reward is over all Available Models) `return (1+(MaxPhases/2.0));`
	Firewall->FilterRequests_case3
	(Reward is over all Available Models) `return 1;`
Reward Function	*(Reward is over all Available Models)*

Simulator Statistics	Type	Time Averaged Interval of Time	
	Options	Estimate Mean	
		Include Lower Bound on Interval Estimate	
		Include Upper Bound on Interval Estimate	
		Estimate out of Range Probabilities	
		Confidence Level is Relative	
	Parameters	Start Time	0.0
		Stop Time	30.0
	Confidence	Confidence Level	0.95
		Confidence Interval	0.1

Performance Variable : fracCorruptServers

Affecting Models	Server	
Impulse Functions		

Reward Function	*(Reward is over all Available Models)*
	`if (Server->NumActive->Mark() > 0)`
	`          return Server->NumCorrupt->Mark()*1.0/(Server->NumActive-`
	`>Mark()*NumServers);`	

Simulator	Type	Instant of Time

Statistics	Options	Estimate Mean	
		Include Lower Bound on Interval Estimate	
		Include Upper Bound on Interval Estimate	
		Estimate out of Range Probabilities	
		Confidence Level is Relative	
	Parameters	Start Time	30.0
	Confidence	Confidence Level	0.95
		Confidence Interval	0.1

Performance Variable : StrongUnavailability

Affecting Models	Server		
Impulse Functions			

Reward Function	*(Reward is over all Available Models)*
	`if(Server->NumCorrupt->Mark()>0 || Server->NumActive->Mark()==0)`
	`        return 1.0/NumServers;`		

Simulator Statistics	Type	Time Averaged Interval of Time	
	Options	Estimate Mean	
		Include Lower Bound on Interval Estimate	
		Include Upper Bound on Interval Estimate	
		Estimate out of Range Probabilities	
		Confidence Level is Relative	
	Parameters	Start Time	0.0
		Stop Time	30.0
	Confidence	Confidence Level	0.95
		Confidence Interval	0.1

Performance Variable : WeakUnavailability

Affecting Models	Server		
Impulse Functions			

Reward Function	*(Reward is over all Available Models)*
	`if (Server->NumActive->Mark() == Server->NumCorrupt->Mark())`
	`        return 1.0/NumServers;`		

Simulator Statistics	Type	Time Averaged Interval of Time	
	Options	Estimate Mean	
		Include Lower Bound on Interval Estimate	
		Include Upper Bound on Interval Estimate	
		Estimate out of Range Probabilities	
		Confidence Level is Relative	
	Parameters	Start Time	0.0
		Stop Time	30.0
	Confidence	Confidence Level	0.95
		Confidence Interval	0.1

Performance Variable : numCorrupt			
Affecting Models	Server		
Impulse Functions			
Reward Function	*(Reward is over all Available Models)* `return (Server->Corruption->Mark()>=MaxPhases);`		
Simulator Statistics	Type	Instant of Time	
	Options	Estimate Mean	
		Include Lower Bound on Interval Estimate	
		Include Upper Bound on Interval Estimate	
		Estimate out of Range Probabilities	
		Confidence Level is Relative	
	Parameters	Start Time	30.0
	Confidence	Confidence Level	0.95
		Confidence Interval	0.1

Performance Variable : goodthroughput			
Affecting Models	Server		
Impulse Functions	Server->ServeReq_case4		

	(Reward is over all Available Models)
	`if (Server->Corruption->Mark() < MaxPhases)`
	`        return 1;`		

Reward Function	*(Reward is over all Available Models)*		
Simulator Statistics	Type	Time Averaged Interval of Time	
	Options	Estimate Mean	
		Include Lower Bound on Interval Estimate	
		Include Upper Bound on Interval Estimate	
		Estimate out of Range Probabilities	
		Confidence Level is Relative	
	Parameters	Start Time	0.0
		Stop Time	30.0
	Confidence	Confidence Level	0.95
		Confidence Interval	0.1

Performance Variable : badthroughput	
Affecting Models	Server
Impulse Functions	Server->ServeReq_case4

	(Reward is over all Available Models)
	`if (Server->Corruption->Mark() == MaxPhases)`
	`        return 1;`

Reward Function	*(Reward is over all Available Models)*

Simulator Statistics	Type	Time Averaged Interval of Time	
	Options	Estimate Mean	
		Include Lower Bound on Interval Estimate	
		Include Upper Bound on Interval Estimate	
		Estimate out of Range Probabilities	
		Confidence Level is Relative	
	Parameters	Start Time	0.0
		Stop Time	30.0
	Confidence	Confidence Level	0.95
		Confidence Interval	0.1

Performance Variable Model: MRDM_PV_SS		
Top Level Model Information	Child Model Name	MRDM
	Model Type	Rep/Join

Performance Variable : SSfracCorruptServers			
Affecting Models	Server		
Impulse Functions			

Reward Function	*(Reward is over all Available Models)*
	`if (Server->NumActive->Mark() > 0)`
	`        return Server->NumCorrupt->Mark()*1.0/(Server->NumActive->Mark()*NumServers);`		

Simulator Statistics	Type	Steady State	
	Options	Estimate Mean	
		Include Lower Bound on Interval Estimate	
		Include Upper Bound on Interval Estimate	
		Estimate out of Range Probabilities	
		Confidence Level is Relative	
	Parameters	Initial Transient	5.0
		Batch Size	1000.0
	Confidence	Confidence Level	0.95
		Confidence Interval	0.01

Performance Variable : SSStrongUnavailability		
Affecting Models	Server	
Impulse Functions		

Reward Function	*(Reward is over all Available Models)*
	`if(Server->NumCorrupt->Mark()>0 || Server->NumActive->Mark()==0)`
	`        return 1.0/NumServers;`	

Simulator Statistics	Type	Steady State
	Options	Estimate Mean
		Include Lower Bound on Interval Estimate

		Include Upper Bound on Interval Estimate	
		Estimate out of Range Probabilities	
		Confidence Level is Relative	
	Parameters	Initial Transient	5.0
		Batch Size	1000.0
	Confidence	Confidence Level	0.95
		Confidence Interval	0.01

Performance Variable : SSWeakUnavailability			
Affecting Models	Server		
Impulse Functions			

Reward Function	*(Reward is over all Available Models)*
	`if (Server->NumActive->Mark() == Server->NumCorrupt->Mark())`
	`        return 1.0/NumServers;`		

Simulator Statistics	Type	Steady State	
	Options	Estimate Mean	
		Include Lower Bound on Interval Estimate	
		Include Upper Bound on Interval Estimate	
		Estimate out of Range Probabilities	
		Confidence Level is Relative	
	Parameters	Initial Transient	5.0
		Batch Size	1000.0
	Confidence	Confidence Level	0.95
		Confidence Interval	0.01

Performance Variable : SSnumCorrupt			
Affecting Models	Server		
Impulse Functions			
Reward Function	*(Reward is over all Available Models)* `return (Server->Corruption->Mark()>=MaxPhases);`		
Simulator Statistics	Type	Steady State	
	Options	Estimate Mean	
		Include Lower Bound on Interval Estimate	
		Include Upper Bound on Interval Estimate	
		Estimate out of Range Probabilities	
		Confidence Level is Relative	
	Parameters	Initial Transient	5.0
		Batch Size	1000.0
	Confidence	Confidence Level	0.95
		Confidence Interval	0.01

Performance Variable : SSgoodthroughput			
Affecting Models	Server		
Impulse Functions	Server->ServeReq_case4		
	(Reward is over all Available Models)		

	`if (Server->Corruption->Mark() < MaxPhases)`
	`        return 1;`		

Reward Function	*(Reward is over all Available Models)*		
Simulator Statistics	Type	Steady State	
	Options	Estimate Mean	
		Include Lower Bound on Interval Estimate	
		Include Upper Bound on Interval Estimate	
		Estimate out of Range Probabilities	
		Confidence Level is Relative	
	Parameters	Initial Transient	5.0
		Batch Size	1000.0
	Confidence	Confidence Level	0.95
		Confidence Interval	0.01

Performance Variable : SSbadthroughput			
Affecting Models	Server		
Impulse Functions	Server->ServeReq_case4		
	(Reward is over all Available Models)		

	`if (Server->Corruption->Mark() == MaxPhases)`
	`        return 1;`		

Reward Function	*(Reward is over all Available Models)*		
Simulator Statistics	Type	Steady State	
	Options	Estimate Mean	
		Include Lower Bound on Interval Estimate	
		Include Upper Bound on Interval Estimate	
		Estimate out of Range Probabilities	
		Confidence Level is Relative	
	Parameters	Initial Transient	5.0
		Batch Size	1000.0
	Confidence	Confidence Level	0.95
		Confidence Interval	0.01

Range Study Variable Assignments for Study *Detection* in Project *MRDM* :

Variable	Type	Range Type	Range	Increment	Increment Type	Function	n
AgreementDelay	double	Fixed	.15	-	-	-	-
DetectionRate	double	Fixed	1	-	-	-	-
FilterRate	double	Fixed	10	-	-	-	-
IncrementalDelay	double	Fixed	.01	-	-	-	-
MaxGwQLen	short	Fixed	10	-	-	-	-
MaxPhases	short	Fixed	5	-	-	-	-
NumServers	short	Fixed	12	-	-	-	-
ProbDetection	double	Manual	[0, .1, .2, .3, .4, .6, .8, 1]	-	-	-	-
ProbFiltering	double	Fixed	0.4	-	-	-	-
ProbMultiPhase	double	Fixed	0.001	-	-	-	-
ProbSinglePhase	double	Fixed	.002	-	-	-	-
RepairRate	double	Fixed	.06	-	-	-	-
ServiceRate	double	Fixed	16	-	-	-	-
fracPhases	double	Fixed	.6	-	-	-	-

Range Study Variable Assignments for Study *MultiPhase* in Project *MRDM* :

Variable	Type	Range Type	Range	Increment	Increment Type	Function	n
AgreementDelay	double	Fixed	.15	-	-	-	-
DetectionRate	double	Fixed	1	-	-	-	-
FilterRate	double	Fixed	10	-	-	-	-
IncrementalDelay	double	Fixed	.01	-	-	-	-
MaxGwQLen	short	Fixed	10	-	-	-	-
MaxPhases	short	Fixed	5	-	-	-	-
NumServers	short	Fixed	12	-	-	-	-
ProbDetection	double	Fixed	0.5	-	-	-	-
ProbFiltering	double	Fixed	0.4	-	-	-	-
ProbMultiPhase	double	Incremental	[0.0,0.009000000000000001]	.001	Additive	-	-
ProbSinglePhase	double	Fixed	0	-	-	-	-
RepairRate	double	Fixed	.06	-	-	-	-
ServiceRate	double	Fixed	16	-	-	-	-
fracPhases	double	Fixed	.6	-	-	-	-

Range Study Variable Assignments for Study *NumServers* in Project *MRDM* :

Variable	Type	Range Type	Range	Increment	Increment Type	Function	n
AgreementDelay	double	Fixed	.15	-	-	-	-
DetectionRate	double	Fixed	1	-	-	-	-
FilterRate	double	Fixed	10	-	-	-	-
IncrementalDelay	double	Fixed	.01	-	-	-	-
MaxGwQLen	short	Fixed	10	-	-	-	-
MaxPhases	short	Fixed	5	-	-	-	-
NumServers	short	Incremental	[4,13]	3	Additive	-	-
ProbDetection	double	Fixed	0.5	-	-	-	-
ProbFiltering	double	Fixed	0.4	-	-	-	-
ProbMultiPhase	double	Fixed	0.001	-	-	-	-
ProbSinglePhase	double	Fixed	.002	-	-	-	-
RepairRate	double	Fixed	.06	-	-	-	-
ServiceRate	double	Fixed	16	-	-	-	-
fracPhases	double	Fixed	.6	-	-	-	-

Range Study Variable Assignments for Study *MRDM_Study* in Project *MRDM* :

Variable	Type	Range Type	Range	Increment	Increment Type	Function	n
AgreementDelay	double	Fixed	.15	-	-	-	-
DetectionRate	double	Fixed	1	-	-	-	-
FilterRate	double	Fixed	10	-	-	-	-
IncrementalDelay	double	Fixed	.01	-	-	-	-
MaxGwQLen	short	Fixed	10	-	-	-	-
MaxPhases	short	Fixed	5	-	-	-	-
NumServers	short	Fixed	12	-	-	-	-
ProbDetection	double	Fixed	0.5	-	-	-	-
ProbFiltering	double	Fixed	0.4	-	-	-	-
ProbMultiPhase	double	Fixed	0.001	-	-	-	-
ProbSinglePhase	double	Fixed	.002	-	-	-	-
RepairRate	double	Fixed	.06	-	-	-	-
ServiceRate	double	Fixed	16	-	-	-	-
fracPhases	double	Fixed	.6	-	-	-	-

Range Study Variable Assignments for Study *MRDM_Study_SS* in Project *MRDM* :

Variable	Type	Range Type	Range	Increment	Increment Type	Function	n
AgreementDelay	double	Fixed	.15	-	-	-	-
DetectionRate	double	Fixed	1	-	-	-	-
FilterRate	double	Fixed	10	-	-	-	-
IncrementalDelay	double	Fixed	.01	-	-	-	-
MaxGwQLen	short	Fixed	10	-	-	-	-
MaxPhases	short	Fixed	5	-	-	-	-
NumServers	short	Fixed	12	-	-	-	-
ProbDetection	double	Fixed	.5	-	-	-	-
ProbFiltering	double	Fixed	.4	-	-	-	-
ProbMultiPhase	double	Fixed	.001	-	-	-	-
ProbSinglePhase	double	Fixed	.002	-	-	-	-
RepairRate	double	Fixed	.06	-	-	-	-
ServiceRate	double	Fixed	16	-	-	-	-
fracPhases	double	Fixed	.6	-	-	-	-

Range Study Variable Assignments for Study *Repair* in Project *MRDM* :

Variable	Type	Range Type	Range	Increment	Increment Type	Function	n
AgreementDelay	double	Fixed	.15	-	-	-	-
DetectionRate	double	Fixed	1	-	-	-	-
FilterRate	double	Fixed	10	-	-	-	-
IncrementalDelay	double	Fixed	.01	-	-	-	-
MaxGwQLen	short	Fixed	10	-	-	-	-
MaxPhases	short	Fixed	5	-	-	-	-
NumServers	short	Fixed	12	-	-	-	-
ProbDetection	double	Fixed	0.5	-	-	-	-
ProbFiltering	double	Fixed	0.4	-	-	-	-
ProbMultiPhase	double	Fixed	0.001	-	-	-	-
ProbSinglePhase	double	Fixed	.002	-	-	-	-
RepairRate	double	Manual	[0, .01, .02, .03, .04, .05, .1, .15]	-	-	-	-
ServiceRate	double	Fixed	16	-	-	-	-
fracPhases	double	Fixed	.6	-	-	-	-

Range Study Variable Assignments for Study *SinglePhase* in Project *MRDM* :

Variable	Type	Range Type	Range	Increment	Increment Type	Function	n
AgreementDelay	double	Fixed	.15	-	-	-	-
DetectionRate	double	Fixed	1	-	-	-	-
FilterRate	double	Fixed	10	-	-	-	-
IncrementalDelay	double	Fixed	.01	-	-	-	-
MaxGwQLen	short	Fixed	10	-	-	-	-
MaxPhases	short	Fixed	5	-	-	-	-
NumServers	short	Fixed	12	-	-	-	-
ProbDetection	double	Fixed	0.5	-	-	-	-
ProbFiltering	double	Fixed	0.4	-	-	-	-
ProbMultiPhase	double	Fixed	0	-	-	-	-
ProbSinglePhase	double	Incremental	[0.0,0.009000000000000001]	.001	Additive	-	-
RepairRate	double	Fixed	.06	-	-	-	-
ServiceRate	double	Fixed	16	-	-	-	-
fracPhases	double	Fixed	.6	-	-	-	-