Designing SCIT Architecture Pattern in a Cloud-based Environment

Quyen L. Nguyen and Arun Sood
International Cyber Center and Department of Computer Science
George Mason University, Fairfax, VA 22030
{*qnguyeng@gmu.edu, asood@gmu.edu*}

Abstract—**Cloud Computing has gained momentum in the IT world, due to its inherent elasticity that allows flexible on-demand computing resources such as CPU time, memory and storage size. However, Cloud security is a challenge. In this paper, we leverage Cloud services to design C-SCIT (Cloud-based Self-Cleansing Intrusion Tolerant) scheme that can provide enhanced intrusion tolerance to applications and services deployed in the Cloud. Challenges and issues of such an approach are analyzed against the traditional implementation, which relies solely on servers internal to an organization. We will show how we can control and adapt the design to satisfy different levels of intrusion tolerance.**

Keywords – SCIT; Intrusion Tolerance; Cloud Computing, Intercloud.

I. Introduction

Cloud Computing has gained momentum in the IT world, due to its inherent elasticity that allows flexible on-demand computing resources such as CPU time, memory and storage size. Indeed, more and more organizations are moving their applications to the clouds, or use clouds for infrastructure and platform services. But, along with the increasing adoption of this new paradigm with its multi-tenancy of storage and computing resources, come the concerns about security, including data confidentiality, integrity, and availability.

Given the distributed nature of cloud services to be provided over networks and open protocols, there is no guarantee that cloud services are not vulnerable to malicious attacks. This vulnerability persists despite the many prevention and detection measures deployed by cloud vendors to protect their assets and business services. The bad news is that security attacks have become more and more sophisticated. Therefore, a system cannot rely solely on intrusion prevention and detection for its security protection, but should have intrusion tolerance systems (ITS) as part of the solution for the cloud computing environment.

The main contribution of this paper is to design a Cloud-based Self-Cleansing Intrusion Tolerance (C-SCIT), a recovery-based intrusion tolerance scheme leveraging Cloud services from multiple vendors. Since our approach utilizes multiple Cloud providers for computing resources as well as storage resources and enhances the security of applications and services, it could be considered as a service in the Intercloud layer [6]. Challenges and issues of such an approach are compared against the traditional implementation, which relies solely on servers internal to an organization. We will also show how we can control and adapt the design to satisfy different levels of resilient. The current SCIT implementation utilizes redundancy and internet exposure management to enhance system security, Diversity has been recognized to be a potential factor to enhance security, but has not been attempted because of cost considerations. In this regard, C-SCIT exploits the diversity property that is inherent to the Intercloud environment to further augment the resilience of the system's intrusion tolerance.

This paper is organized as follows. Section II provides related research in two areas of Intercloud and recovery-based intrusion tolerance. Section III contains an overview of SCIT architecture [9], which will be augmented in later sections to yield a dependable cloud architecture. System components for a Cloud-based SCIT (C-SCIT) are described in Section IV. C-SCIT procedures are in Section V. In Section VI, we show how C-SCIT can provide desired levels of intrusion tolerance. Section VII illustrates some possible implementations of C-SCIT procedures using commercially available Cloud Services APIs. The paper ends with a conclusion and discussion of future work in Section VIII.

II. Related Work

There are two types of related work. On the Cloud side, the notion of a service in the Intercloud was mentioned in [4,5,6,7,8]. A lot of research has been published on Cloud Storage, in the multi-tenancy environment offered by Cloud Computing. Abu-Libdey et al. [1] proposed a scheme of Redundant Array of Cloud Storage (RACS) to circumvent the issue of vendor lock-in. The proposal can be viewed as an abstraction of RAID (Redundant Array of Independent Disks) principle. With RACS, digital objects are replicated and distributed among multiple Cloud Storage providers such that the objects can be reconstituted based on just a subset of these providers. Therefore, the data owner does not have to rely on any single vendor to retrieve or rebuild the data.

Cachin et al. [7] have developed theories and protocols in the Intercloud to ensure the integrity, and confidentiality of data stored in Clouds.

The concept of Cloud Federation and market-oriented Cloud services exchange was promoted in [6] in order to facilitate buying-and-selling of Cloud services and resources rapidly and maybe systematically.

Bessani et al. [5] proposed to use a synthesis of protocols including Byzantine Fault Tolerance, and secret sharing protocols operated in the "cloud-of-clouds" to solve the problem of Cloud Storage dependability.

On the intrusion tolerance side, there have been schemes based on the recovery mechanism. Sousa et al. [13] proposed a system that optimizes efficiency and survivability by having a hybrid solution combining periodic system rejuvenation with "reactive recovery". This recovery is triggered when the perceived threat goes beyond a tolerable threshold that can affect the correct working of the system. In the same vein, the FOREVER service consists of removing faults via proactive reconfiguration implemented by robust underlying components [14]. Reise et al. [12] reported their implementation of proactive recovery on Hypervisor virtualization. In this paper, we want to extend the traditional SCIT described in [9] to the Cloud environment. To the best of our knowledge, there has not been a recovery-based intrusion tolerance system designed to leverage the characteristics of Cloud Computing.

III. SCIT ARCHITECTURE PATTERN

A. High Level Description.

SCIT architecture has as the goal to protect applications and services against malicious faults. SCIT employs a recovery-based mechanism. This mechanism consists of automatic and periodic cleansing of the servers, applications and services on a virtualized or non-virtualized platform. Not only does the periodic cleansing provide almost continuous clean environment for the servers and applications, but by restricting the exposure window of a server replica, we can limit the success rate of malicious attacks. Major components of SCIT pattern are depicted in Figure 1. At the core is the Central Controller managing and controlling all the SCIT nodes to be protected. The nodes are usually grouped into clusters, each of which contains servers with identical functionalities. Diversity of these servers can be employed to further reduce the likelihood of malicious exploitations. For example, we can have a group of web servers, one running on Linux, another on Windows, and a third one on Mac OS. Each node within a cluster of similar nodes is directed by the Controller to constantly go through the following orderly lifecycle:

- Live Spare state, which is a known good state but offline,
- Active state of duration W_o, where the server becomes online to accept and process incoming requests,
- Grace Period state of duration W_g, where the server stops accepting new transaction requests and completes processing of requests still in the queue,
- Inactive state, where the server is offline and undergoes the restoration cleansing to a known good state.

The time period of the rotation through the four states above depends on the cleansing time, which is specific to an application or service. In Figure 1, *Node 1* is in Active state, *Node 2* in Live Spare state, and *Node n* in Inactive state.

The Controller must be deployed on a physical host separate from the one(s) used to deploy the nodes in order to avoid a data path from the applications back to the Controller, thus any compromise made to the application nodes cannot propagate to the Controller. Furthermore, in the current SCIT pattern, the link from the Controller to the online node is unidirectional. But, the links from the Controller to the offline nodes are bidirectional to allow the Controller to check the status of the offline nodes during the cleansing process. Note that the Controller has full control over switching these links from one-way to two-way and vice-versa.

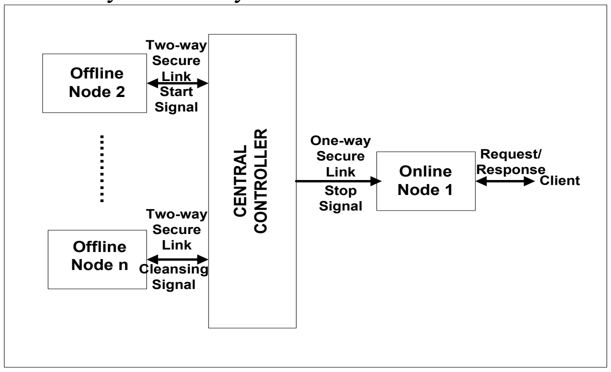


Figure 1. SCIT Architecture Components.

B. SCIT Parameters

Mean Time to Security Failure. In our previous work [10], we have derived a lower bound for the Mean Time to Security Failure (MTTSF) as shown in expression (1) below:

$MTTSF \geq F_{\lambda\,\mu}(W_o)$, where

(1).

In (1), $h=(h_0, h_1, h_2)$ is the vector composed of mean sojourn times in each of the states - Good, Vulnerable, and Attacked respectively; λ is the attack rate and μ is the average residence time in the server. It is clear from (1) that MTTSF can be made to depend solely on the online window W_o. Moreover, we have shown in [10] that as the online window decreases, the intrusion tolerance expressed by MTTSF increases, i.e. the system is more intrusion tolerant.

S-Reliability of service or an application was introduced in [11] to characterize the quality of service, i.e. application functions reliably without the impact of malicious attacks.

IV. CLOUD-BASED APPROACH

In the previous section, we presented the SCIT mechanism that provides intrusion tolerance to applications and services. If we abstract out that mechanism to the level of architecture for recovery-based intrusion tolerance, then we can design and implement the architecture by extending it to Cloud-based environment.

A. System Overview

A C-SCIT system has four major entities: Central Controller, Proxy, SCIT Local Controllers, and Application Nodes.

The *Application Nodes* are application replicas that have to be protected by C-SCIT, such as Web Server, or Web Application. Each Node is assigned to run in a domain on the computing platforms of the Cloud Providers. We can envision having one Node per Cloud, and the number of Nodes depends on the level of intrusion tolerance required by the user's application. The clean image of the Application Node must be safely stored locally in the Cloud Storage. We have a copy of the image in each Cloud instead of a centralized location so as to have it close to the Application Node, thus avoiding large data transfer for each cleansing cycle.

Each Application Node is managed by a *SCIT Local Controller*. A Local Controller is responsible for maintaining the clean image for the Node, and directing its associated Node through the four states of Life Spare, Active, Grace Period, and Inactive. Maintaining the image includes keeping it in a protected area, along with the capability to compute the checksum of the image based on the algorithm dictated by the Central Controller. The link between an Application Node and a SCIT Local Controller is unidirectional from the latter to the former so that any compromise to the Application cannot affect the Local Controller. The existence of Local Controllers aims at limiting the amount of messages and data to be exchanged between the Nodes and the Central Controller over the WAN during C-SCIT operations.

The *Central Controller* coordinates the cleansing procedure by initiating an operational message to the SCIT Local Controllers running in a virtualized platform at each Cloud Service Provider. This control includes the assurance that the images of the Application/Service instances are pristine. Preferably, the Central Controller is deployed in an environment/domain outside of the Cloud environments where the Application Instances and SCIT Local Controllers reside. The Controller's domain can be on the customer's premise or a third-party Cloud.

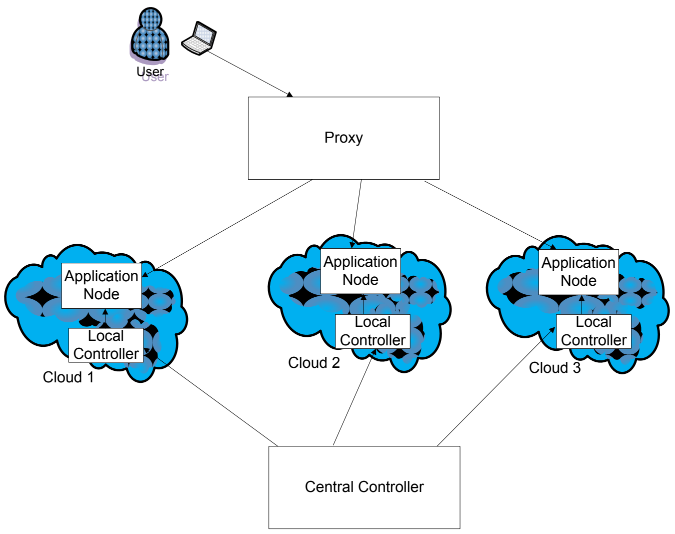


Figure 2. System Architecture of C-SCIT.

Finally, the *Proxy* has the task of interfacing directly with application end-users and mediating requests and responses between the end-users and the active Application Node. Note that an end-user can be a human or another system. It could also be a service participating in a composite application in a Service-Oriented Architecture (SOA) system. In Figure 2, the unidirectional link between the Central Controller and the Proxy was not shown to avoid making the figure too busy. It is clear that the design of C-SCIT added the new entity of SCIT Local Controller and the delegation of some of the duties from the Central Controller to the Local Controllers.

B. *Central Controller Components*

In order to manage the set of SCIT Local Controllers and indirectly the Application Nodes, the Central Controller needs the following components:

- Service Registry storing information about the applications and services to be managed and protected, including their images with checksums, their IT-QoS requirements, and operational parameters such as the exposure windows.
- Algorithm Modules computing the operational parameters based on the desired levels of intrusion tolerance.

C. *Advantages*

While C-SCIT fulfills the self-cleansing as specified by the SCIT architecture, C-SCIT brings additional advantages in terms of dependability and intrusion tolerance thanks to the characteristics of the Cloud Computing paradigm.

Geographical Distribution. Since the Application Nodes are dispersed to different Cloud vendors, whose data centers are most likely to be geographically distributed, C-SCIT benefits from this feature, which conceptually enhances its dependability, especially against natural disaster, power outage, or weather storm. Actually, even some individual Cloud providers such as Amazon have offered the option to allow an application to deliver content from different geographical sites [2].

Diversity. Another benefit is that C-SCIT enjoys some diversity from having Nodes among various Clouds. Indeed, it is probable that two Clouds would have dissimilar computing platforms, including OS and virtual machine products, so that one can always select vendors with diverse environments. Speculatively, this diversity may decrease the success rate of malicious attacks, except in the case of a concerted DDoS attack.

Flexible Resource Utilization. Assuming that some Service Level Agreement (SLA) has been negotiated with Cloud vendors for C-SCIT, flexibility and "pay-as-you-go" model for resource utilization are useful for C-SCIT in its adaptive support for satisfying Intrusion Tolerance QoS (IT-QoS), as we will show in a later section. The notion of IT-QoS was introduced in [11] to help the design and implementation of intrusion tolerance characteristics for services.

D. *Challenges*

Given the benefits discussed above, what are the challenges in realizing C-SCIT due to Cloud-based environment?

Connection Security. SCIT pattern requires that the communication between Central Controller and Nodes be trusted to prevent any breach of the Controller. This can be achieved easily if the Controller and Nodes are collocated and within the confine of a well protected network. Another option to use a hardware solution was described in [9]. In the case of C-SCIT, the connection between a SCIT Local Controller and its managed Node can be made secure since they are also co-located within one Cloud's data center. For the connection between the Central Controller and the SCIT Local Controllers, a secure Virtual Private Network link must be established to ensure that signals and messages cannot be tampered.

Connection Reliability. The other challenge of C-SCIT is that connection between the Central Controller and the SCIT Local Controllers communicate with each other over a network connection, most likely a WAN, which is susceptible to failure. This failure will affect the requirement that C-SCIT has to provide some guarantee with respect to MTTSF (Mean Time to Security Failure), availability and S-Reliability. In this paper, we assume that there is an SLA specifying the connection reliability. Without this assumption, retry protocol needs to be devised to respond to this challenge.

Image Integrity. Since the cleansing procedure depends on the application image, we need a scheme to ensure its integrity. This image will change over time, due to patches and upgrades made to the application. Image updates will be triggered by the Central Controller sending a signal to every SCIT Local Controller managing the same application. The Central Controller keeps the checksum of the application's image in its "safe store", which is part of the Service Registry.

V. C-SCIT PROCEDURES

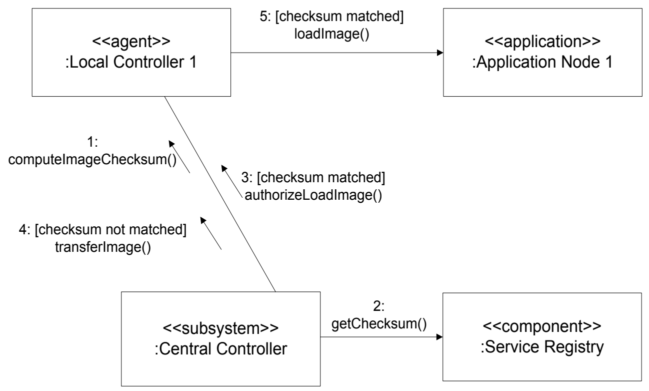
C-SCIT has two main procedures to perform self-cleansing of the applications and services. The cleansing procedure is applied to an Application Node after its Grace Period expires. The Activation procedure will switch the Live-Spare Node to the Online mode.

A. Assumptions

We made two assumptions: a) The Service Registry is a trusted repository of all the application images; b) The connection link between the Central Controller and the Local Controllers in the Clouds is reliable. If assumption a) is not valid as in the case where the Controller and its components are deployed in a separate Cloud, then a more complex scheme has to be designed. Should assumption b) fail to hold, either a retry scheme or a scheme that switches the cleansing procedure to another Application Node has to be instituted.

B. Cleansing Procedure

The procedure described below assumes that the Service Registry is fully trusted, only accessible by the Central Controller, and maintains the authoritative image copy of the application/service (Figure 3). Before authorizing the SCIT Local Controller to load the pristine image on the Application Node, the Central Controller verifies whether the checksum computed by the Local Controller matches with the one stored in the Service Registry (steps 1-2). If the two checksums match (step 3), then the Central Controller gives a go ahead with loading the image via authorizeLoadImage(). If the two checksums don't match, the Central Controller will transfer a new image using transferImage() (step 4), then repeat the checksum verification process as above. Verifying the checksum of the image stored locally in the Cloud Storage of the Application Node is necessary to make sure of the image's integrity before the loading.



<<agent>> :Local Controller 1
5: [checksum matched] loadImage()
<<application>> :Application Node 1
1: computeImageChecksum()
3: [checksum matched] authorizeLoadImage()
4: [checksum not matched] transferImage()
<<subsystem>> :Central Controller
2: getChecksum()
<<component>> :Service Registry

Figure 3. C-SCIT Cleansing Procedure Diagram.

C. Activation Procedure

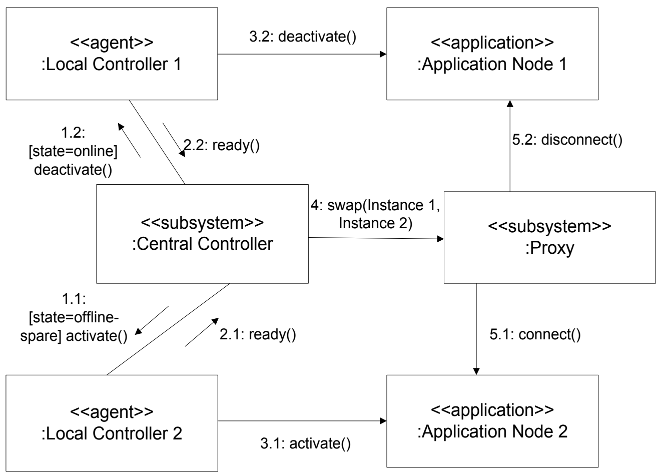
The Activation procedure in C-SCIT shown in Figure 4 below consists of the following steps:

<u>Step 1</u>. When the time comes to initiate the procedure, the Central Controller sends two messages in parallel to two SCIT Local Controllers: a deactivate() message to the Local Controller managing the currently active Application Node, and an activate() message to the one associated with the live-spare Application Node. In the above diagram, the active pair is (SCIT Local Controller 1, Application Node 1) and the live-spare one is (SCIT Local Controller 2, Application Node 2).

<u>Step 2</u>. The SCIT Local Controllers turn around and sends the appropriate orders to their managed Nodes. In the above example, SCIT Local Controller 1 deactivates Node 1, while SCIT Local Controller 2 activates Node 2.

<u>Step 3</u>. Upon receiving the ready() messages from both Nodes, the Central Controller requests the Proxy to replace the currently active Node with the newly active Node.

Note that at the beginning when there is no active Node, there is only one step that is to activate the first Live Spare Node.



<<agent>> :Local Controller 1
3.2: deactivate()
<<application>> :Application Node 1
1.2: [state=online] deactivate()
2.2: ready()
5.2: disconnect()
<<subsystem>> :Central Controller
4: swap(Instance 1, Instance 2)
<<subsystem>> :Proxy
1.1: [state=offline-spare] activate()
2.1: ready()
5.1: connect()
<<agent>> :Local Controller 2
3.1: activate()
<<application>> :Application Node 2

Figure 4. C-SCIT Activation Procedure Diagram.

VI. Providing IT-QoS Levels

A. Impact on SCIT Parameters

Unlike in the implementation where Controller and Nodes are co-located in a data center, the distributed nature of C-SCIT will incur some network latency. So, this latency has to be factored in the evaluation of the cleansing time $T_{cleansing}$.

Let C = {C_1, …, C_p} be the set of available Clouds. For the purpose of C-SCIT, each Cloud C_i is characterized by the image loading time (t_{load}), communication latency (t_{com}), checksum verification time (t_{verify}), probability that checksum test fails (P_{cf}), and maybe correction time ($t_{correct}$). The latter is incurred only when the image stored for the to-be-cleansed Node fails the checksum test, and the image needs to be re-transferred to the problematic Cloud as shown in step 4 of Figure 4. Thus, we can write:

$$C_i = (t_{i,load}, t_{i,com}, t_{i,verify}, t_{i,correct}, P_{i,cf}) \qquad (2).$$

Cleansing Time. Given the above procedure depicted in Figure 3, the average cleansing time for an Application Node in C-SCIT is:

$$t_{i,cleansing} = t_{i,load} + t_{i,com} + t_{i,verify} + P_{i,cf} * t_{i,correct} \qquad (3).$$

Then, the overall cleansing time for C-SCIT system becomes:

$$T_{cleansing} = \max \{ t_{i,cleansing} ; \text{for } i = 1, .., n\}. \qquad (4).$$

Note that the complexity degree of evaluating $T_{correct}$ is a function of the complexity of the correction scheme, especially when assumptions about Service Registry's security and connection link's reliability mentioned in the Cleansing Procedure section don't hold.

Activation Time. From Figure 4, we need to define the time spent by the Activation process, as it involves a few messages. Normally, this activation time is small compared to $T_{cleansing.}$ The overall activation time will be the maximum of all activation times:

$$T_{activation} = \max \{ t_{i,activation} \} \text{ for } i = 1, .., n. \qquad (5).$$

Number of Pairs. More pairs (Local Controller, Application Node) would be required as:

- The smaller the exposure window W_o is, the faster is the rotation cycle;
- The cleansing time $T_{cleansing}$ has an adverse effect on the rotation cycle, i.e. the longer it takes to restore a node, the longer is the rotation cycle.

Based on this, we arrive at the expression for the number of Nodes N that is needed to provide an exposure window of value W_o:

———————— (6).

If we let $W_g = W_o$, then (2) becomes:

——————— 2 (7).

For example, if the average $T_{cleansing}$ is equal to 12 min and $T_{activation}$ = 2 min, and we want W_o = 3 min in order to achieve a certain level for MTTSF, then the C-SCIT Intercloud requires N = 7 pairs. Unlike the enterprise environment where the maximum of Application Nodes is constrained by the hardware configuration of the physical hosts that have been procured, the Cloud environment offers much more elasticity in terms spawning more or less virtual machines for Application Nodes and Local Controllers.

B. C-SCIT Operation

In this section, we will examine how C-SCIT operates to satisfy a desired MTTSF for an application or service. Based on the analytical results obtained in [13] that MTTSF can be tuned by varying the exposure window W_0, the function **calculateConfig**() returns W_0 and the number of Nodes necessary to maintain the given level of MTTSF. Figure 5 summarizes the main steps of C-SCIT operational activity embodied in the function **C-SCIT-Operate()**. In the scheme presented below, C denotes the set of Clouds with which C-SCIT has established SLAs. The queue Q_g is built out of a subset of C. At one point in time, only the Clouds in Q_g are utilized by the Central Controller. Queue Q_s contains the Clouds with the pair of loaded Application Node and Local Controller. First, the scheme calculates the number of Nodes N and value of exposure window W_o (line 1). Then, N Clouds are reserved in Q_g for C-SCIT operation (line 2). Two threads for cleansing and activation will be instantiated (lines 3-4); they follow the procedures described in section V-A and V-B respectively. Two different timers are used to signal these two threads for when to initiate the actions of cleansing and activation. The values of these timers take into account $T_{cleansing}$ and $T_{activation}$.

```
C-SCIT-Operate()
Input
•   m: MTTSF value;
•   C={C_1,..,C_p}: set of available Clouds;
•   λ and μ: average rate of attacks and attack residence
    respectively;

1.  (W_0, N) ← calculateConfig(m, C);
2.  Build queue Q_g with first N Clouds taken from C;
3.  Fork a thread and invoke cleanse(Q_g);
4.  Fork a thread and invoke activate(Q_g);

calculateConfig(m, C)
5.  Compute T_cleansing using (4).
6.  Compute T_activation using (5).
7.  Solve equation F(2w) = m for w, where F is the
    expression given by (1). Let W_sol be the solution.
8.  W_o ← W_sol;
9.  W_g ← W_sol;
10. Compute N using (7);
11. return (W_o, N);

cleanse(Q_g)
12. While TRUE
13.   C_i ← dequeue(Q_g);
14.   Perform cleansing steps of C_i as in section V-A.
15.   enqueue(Q_s, C_i);
16.   Wait(timer1);
17. EndWhile

activate(Q_g)
18. While TRUE
19.   C_a ← currently active Cloud;
```

```
20.     $C_s \leftarrow$ dequeue($Q_s$);
21.     If $C_s$ = null
22.       return error;
23.     EndIf
24.     If $C_a$ = null
25.     Then
26.       Activate $C_s$;
27.     Else
28.       Perform activation steps as in section V-B. This
        includes $C_a$ and $C_s$;
29.     EndIf
30.     Wait(timer2);
31.  EndWhile
```

Figure 5. C-SCIT Operation Scheme.

When there is a need to increase or decrease the desired MTTSF, the above scheme can be augmented to reconfigure C-SCIT. For instance, if there is an alert that the attack rate is increasing, and more intrusion intolerance is required, the reconfiguration scheme can be invoked. Indeed, the augmented scheme consists of killing the current threads for cleansing and activating, and call C-SCIT-Operate(). This will trigger a recalculation in line 1, and spawn new threads with the newly computed values for number of Nodes and exposure window. Since Cloud-based environment can potentially and elastically provision largely variable number of virtual machines and domains, adding more Application Nodes to queue Q_g or releasing unused Clouds should not pose any problem, as in the case of an enterprise data center.

VII. Usage Of Commercial Cloud Service API Sets

As a proof of concept, we surveyed the APIs of Amazon Web Services (AWS) [2] and Windows Azure [15] to determine whether the commercially available Cloud Services provide sufficient tools allowing us to implement the proposed C-SCIT. Study results are promising. For example, the function deactivate() issued by a Local Controller to an Application Node can be implemented by the command *ec2-terminate-instances* or the RESTFul web service *StopInstances* for AWS and the *Delete Hosted Service* for Windows Azure. Bringing up an Application Node via the function activate() can be achieved by invoking the AWS *StartInstances* web service and Microsoft's *Create Hosted Service*. Communication between the Central and Local Controllers can be made secure since Cloud Services provide a system for key management to protect message and data exchange, as we have assumed earlier. A prototype will be planned to be built to demonstrate the validity of our assumptions and the feasibility of our proposed design.

VIII. Conclusion and Future work

Motivated by the increasing adoption of Cloud Computing, and the need to make services intrusion tolerant in the Cloud environment, we have proposed a Cloud-based approach in designing SCIT, a recovery-based intrusion tolerance mechanism. The proposed implementation would provide enhanced availability and resilience of the services deployed in the Cloud. Thanks to inherent nature of Cloud Computing, this approach brings additional benefits of elastic redundancy and diversity to the traditional SCIT architecture. The elasticity allows SCIT to easily adapt to reconfigure itself and adapt to new required levels of intrusion tolerance expressed via values of MTTSF for instance, without being constrained by the fixed hardware configuration as in a pure enterprise environment, while multi-cloud diversity hardened the solution against repeated and/or concerted malicious attacks.

In this paper, we have made the assumption of dependability for the connection between the Central Controller and the partnering Clouds. Enhancements such as retries will need to be made by adding more steps in the Cleansing and Activation procedures to ensure dependable operation of C-SCIT. As the performance of retries may risk hindering the operationability of C-SCIT scheme, experiments will have to be performed to further show that these enhancements are possible.

References

[1] Hussam Abu-Libdeh, Lonnie Princehouse, Hakim Weatherspoon. "RACS: A Case for Cloud Storage Diversity". Proceedings of the 1st ACM symposium on Cloud computing, 2010.

[2] Amazon Elastic Compute Cloud (EC2). http://aws.amazon.com/ec2/.

[3] Michael Armbrust et al. "A View of Cloud Computing". Communications of the ACM, Volume 53, No 4, April 2010.

[4] David Bernstein and Deepak Vij. "Intercloud Security Considerations". 2nd IEEE International Conference on Cloud Computing Technology and Science, 2010.

[5] Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando Andre, and Paulo Sousa. "DepSky: Dependable and Secure Storage in a Cloud-of-Clouds". EuroSys'11, April 10-13, 2011, Salzburg, Austria. http:// www.di.fc.ul.pt/~mpc/pubs/eurosys219-bessani.pdf. [02/28/2011].

[6] Rajkumar Buyya, Rajiv Ranjan, and Rodrigo N. Calheiros." InterCloud: Utility-Oriented Federation of Cloud Computing Environments for Scaling of Application Services". ICA3PP 2010, Part I, LNCS 6081, pp. 13–31, 2010.

[7] Christian Cachin, Robert Haas, and Marko Vukolic. "Dependable Storage in the Intercloud". http://domino.research.ibm.com/library/cyberdig.nsf/papers/630549C46339936C852577C200291E78. [02/27/2011].

[8] R. Buyya, Yeo Chee, and S. Venugopal. "Market-Oriented Cloud Computing: Vision, Hype, and Reality for Delivering IT Services as Computing Utilities". 10th International Conference on High Performance Computing and Communications, 2008. HPCC '08. Sep. 25-27, 2008.

[9] Yih Huang, David Arsenault and Arun Sood. "Closing Cluster Attack Windows Through Server Redundancy and Rotations". Sixth IEEE International Symposium on Cluster Computing and the Grid Workshops, May 16-19, 2006, Singapore.

[10] Quyen Nguyen and Arun Sood. "Quantitative Approach to Tuning of a Time-Based Intrusion-Tolerant System Architecture". WRAITS 2009, Lisbon, Portugal. http://wraits09.di.fc.ul.pt/wraits09paper2.pdf. [05/17/2010].

[11] Quyen Nguyen and Arun Sood. "Realizing S-Reliability for Services via Recovery-driven Intrusion Tolerance Mechanism". 2010 International Conference on Dependable Systems and Networks Workshops (DSN-W), Chicago, Illinois. Jun 28-Jul 1, 2010.

[12] Hans P. Reiser et al. "Hypervisor-Based Efficient Proactive Recovery". *Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems, 2007*.

[13] Paulo Sousa, Alysson Neves Bessani, Miguel Correia, Nuno Ferreira Neves, Paulo Verissimo. "Resilient Intrusion Tolerance through Proactive and Reactive Recovery". *13th IEEE International Symposium on Pacific Rim Dependable Computing*, 2007.

[14] Paulo Sousa et al. "The FOREVER Service for Fault/Intrusion Removal". *WRAITS 2008, Glasgow, Scotland*.

[15] Windows Azure. http://www.microsoft.com/en-us/cloud/developer/resource.aspx?resourceId=introducing-windows-azure. [03/11/2011]