Autonomic Self-Healing for MANETs

J. Chaudhry[1], **Y. Lee**[2], **K. Pence**[3], **J. Sztipanovits**[2]
[1]Università degli Studi di Trento, TN, Italy
[2]ISIS, Vanderbilt University, Nashville, TN, USA
[3]EECS-EngM, Vanderbilt University, Nashville, TN, USA

Abstract – *Self-healing systems are considered as cognation-enabled sub form of fault tolerance system. But our experiments that we report in this paper show that self-healing systems can be used for performance optimization, configuration management, access control management and many other functions. The exponential complexity that results from interaction between autonomic systems and users (software and human users) has hindered the deployment and user of intelligent systems for some time. We show that if exceptional complexity is converted into self-growing knowledge, (policies in our case), can make up for the initial development cost of building an intelligent system. In this paper, we propose that AHSEN (Autonomic Healing-based Self management Engine) clearly demarcates the logical ambiguities in contemporary designs and shows its performance through empirical results obtained through experiments.*

Keywords: Automatic Self-Healing, MANETs, Congestion Control

1 Introduction

As the complexity and size of networks increase so do the costs of network management [1]. The preemptive measures have done little to cut down on network management cost. Hybrid networks cater with high levels of Quality of Service (QoS), scalability, and dynamic service delivery requirements. The amplified utilization of hybrid networks i.e. ubiquitous-Zone based (u-Zone) networks has raised the importance of human resources, down-time, and user training costs [10]. The u-Zone networks are the fusion of the cluster of hybrid Mobile Ad-hoc NETworks (MANETs) and high speed mesh network backbones. They provide robust wireless connectivity to heterogeneous wireless devices and take less setup time. The clusters of hybrid networks feature heterogeneity, mobility, dynamic topologies, limited physical security, and limited survivability [2] and the mesh networks provide the high speed feedback to the connected clusters. The applications of MANETs vary in a great range from disaster and emergency response, to entertainment and internet connectivity to mobile users.

Autonomic Computing provides a cheaper solution for robust network management in u-Zone networks in the form of self-management. Self Management is a tool through which performance of the computer systems can be optimized without human user intervention. In [24] Turing et. al. suggests that autonomic systems have exponential complexity which can hamper the appropriate problem marking and also raises the software cost. So it is critical to provide incremental, low cost and time efficient solutions along with minimizing the maintenance cost of the software.

The u-Zone networks contain a highly vast variety of devices connected to them. It is not apposite to address the problems of each category of devices individually. We need to have some general solutions that could entertain a certain set of devices. Moreover, the probability of a management solution made for one type of client would be appropriate for another client is very low. The authors in [17] target the self management in hybrid environment through a '*divide and conquer*' approach by using component-based programming. They propose to rapidly divide the problem into sub-domain and each domain is then assigned 'sub solutions'. The amalgamation of all 'sub solutions' gives the final management solution to the client.

Several network management solutions proposed in [4] [5] [6] [7] are confined strictly to their respective domains i.e. either mesh network or MANETs. A self-management architecture is proposed in [3] for u-zone networks. We have identified the following questions that are still to be answered since there is only a limited amount of published work on the topic:

1. If self-healing is one of the FCAPS functions (Fault, Configuration, Accounting/Administration, Performance, Security) than what is the physical location of self-healing functions whether it should reside on the gateway or at the client end?
2. How does the control, information etc flow from one function to another? Especially how do Self-healing functions interact with the other functions?
3. What are the calling signatures of self-healing functions? If Self-healing functions are fault-removing functions, than what are the functions of Fault Management functions?
4. Are these sub-functions functionally independent? If yes, then there is an evidence of lot of redundancy and if not then how self-healing can be thought of an independent

entity in other words what is the true functionality definition of self-healing?

5. What if the self-management entity itself faces management problem, how should they be tackled?

The questions posted above, encourages us to propose a flexible self-healing architecture [9] that can not only define the individual functionality of the participating management functions but also be lightweight for thin clients. In this paper, we propose a flexible, autonomic, self-management architecture for u-Zone networks that suits clients with varying processing capabilities. We propose that the Context Awareness and Self Optimization should be an 'always-on' function whereas the other management functions should be 'on-demand' e.g. Self Configuration, Fault Management etc. This categorization of functions equally distributes the self- management framework among the clients without being a big liability on a software platform of clients. The clients in a u-zone network contain a high level of diversity so the management solutions should be exclusive. In real time scenarios it is not lucrative to exclusively tailor solutions for each client. For this reason, we propose that almost all the on-demand functions should be composed dynamically using dynamic component integration [15]. This integration can be very time consuming for the clients so transaction thrashing can take place. The dynamic component integration for the on-demand functions needs the mechanism for preventing the thrashing [23]. We use the delay time-based peak load control scheme in order to preventing the transaction trashing caused by an enormous number of service requests in u-Zone-based hybrid networks. The worker pattern [8], connecter-accepter model [14], reactive [13] and proactive [22] approaches are effective in combination but these are not cost effective in real life applications. The use of a Peak Load Control (PLC) mechanism manages the service requests at the gateway. Depending upon the load on the host gateway, the service requests are routed to the peer gateways to eliminate the inconsistency and redundancy at service level. We show the simulation result for proving the stability of performance, According to our experimental result, the proposed delay time algorithm can stably control the heavy overload after the saturation point has been reached and has significant effect on controlling peak loads.

In section 2 we compare our scheme with some of the contemporary solutions proposed. The proposed scheme follows in section 3. An application scenario is discussed in section 4. The implementation details and simulation results are furnished in section 5. This paper ends with a conclusion and discussion of future work.

2 Related Work

In this section we compare our research with the related work. The Robust Self-configuring Embedded Systems (*RoSES*) project [11] aims to target the management faults using self-configuration. It uses graceful degradation as the means to achieve a dependable system. In [12] the authors propose that there are certain faults that can not be removed through configuring of the system, which means that RoSES does not fulfill the definition of self management as proposed in [16]. The *HYWINMARC* [3] uses cluster heads to manage the clusters at local level but does not explain the criteria of their selection. The specifications of Mobile Code Execution Environment (*MCEE*) are absent. Moreover the use of intelligent agents can gives similar results as discussed above in the case of [14] and [15]. To enforce the management at local level, the participating nodes should have some management liberty. However *HYWINMARC* fails to answer the questions rose in the previous section. The Service Synthesizer on the Net (STONE) project [12] explores new possibilities for users to accomplish their tasks seamlessly and ubiquitously. The project focuses on the development of context-aware services in which applications are able to change their functionality depending on the dynamically changing user context.

3 Proposed Architecture

In hybrid wireless networks, the network contains high node density, scale, variable topology and hence mobility issues. At node level, constraints like high hardware cost, almost all nodes being powered by battery, low processing speed and small memory size, limited transmission range, low bandwidth rates, and power scarce are worth mentioning. This variety of features, constraints and capabilities pose a greater hindrance when proposing a comprehensive management framework that could accommodate most types of nodes. The most desired characteristics of such architecture would be that it should be lightweight and could expand its functionality dynamically. This feature is missing in the related literature.

In this section we present the Autonomic Healing-based Self management Engine (AHSEN). Figure 1 shows the client and gateway self-management software architectures. In a gateway assisted environment, it serves to our advantage to do the bulk of the processing e.g. fault analysis, solution composition etc. at the gateway level and let the client do more important things like self monitoring, optimization and local self management using the Normal Functionality Model (NFM). When a mobile device comes in the range of a gateway, upon configuration request, it is provided with a Normal Functionality Model (NFM), which is specially designed for certain device classes. The NFM runs a self check and reports back to gateway SMF that issues a client version of AHSEN. The client AHSEN consists of two directory services for services and plug-ins, client SMF is responsible mainly to host services, plug-ins, and NFM.

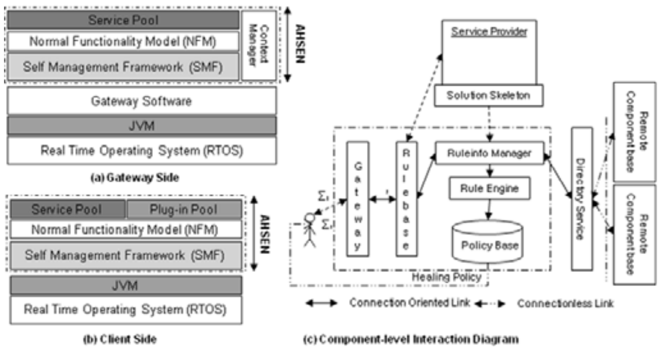

Figure 1: The AHSEN architecture of Client (a) and gateway (b), and Component level Interaction Diagram (c)

The plug-in and service pool contains the directory information of related services provided by different service providers. At client side, the plug-in manager hosts the plug-ins downloaded from remote locations. At server side it contains the directory service that contains the plug-in information. The Normal Functionality Model (NFM) is a device dependent ontology that is downloaded, along with SMF, to the device at the network configuration level. It provides a mobile user with an initial default profile at gateway level plus device level functionality control at the user level. The NFM contains specifications of the normal range of functional parameters of the device, services environment, security certificates and network standards. The SMF constantly traps the user activities and sends them to the SMF at the gateway while hosting the executables. The SMF at gateway directs the trap requests to the context manager who updates the related profile of the user.

Besides the periodic context update ($\sum_i$) the client SMF also sends anomaly report to server SMF ($\sum_j$) which is forwarded to the rule-base for case-based reasoning. If the match is found, its related healing policy is forwarded to the mobile device. In case the problem which is reported by the client SMF, the heuristics are used to determine the cause and solution for the problem reported. The Rule-info Manager collects the results from the rule-base, solution skeleton (which is a policy guideline provided by a third party vendor for problem resolution) and matches it against the components information listed in the directory service using the scheme proposed in. The rule engine composes the components together and generates a healing policy which is stored in the policy base and eventually forwarded to the mobile device. This way, the self managing software grows its knowledge repository. The benefit of using the rule engine is that for every condition there can be several execution components. When the condition part is prepared in the rule engine, a part of it is sent to the rule base. By doing this we increase the rule repository and increase the probability of the fault being detected. The rest of the policy is stored in the policy repository for the mobile device's direct use and the condition part serves an extra job of fault detection in the rule-base.

A detailed architecture of the Self Management Framework (SMF) is divided into two parts: Analyzer, Load Manager. The Analyzer is consisted of The Root Cause Analyzer, RCF Manager, Scheduler, Binding. The man parts of Load Manager are the Traffic Manager, Worker Manager. The Root Cause Analyzer is the core component of the problem detection phase of healing. The State Transition Analysis based approaches [19] might not be appropriate as Hidden Markov Models (HMMs) take long training times along with exhaustive system resource utilization. The profile based Root Cause Detection might not be appropriate mainly because of the vast domain of errors expected [20, 21]. Considering this situation, we use the meta-data obtained from NFM [10] to trigger the Finite State Automata (FSA) series present at the Root Cause Analyzer. In the future we plan to modify the State Transition Analysis Tool [19] in lines of on fault analysis domains. After analyzing the root-cause results from the Root Cause Analyzer (RCA), the Root Cause Fragmentation (RCF) manager in cooperation with the Signature Repository and Scheduler search for the already developed solutions else it arranges a time slot based scheduler for plug-ins. The RCF Manager uses heuristics and gathers all the possible problem causes and their solutions from the already given rule base and hands the context to the scheduler. The scheduler uses the time slot-based mechanism [9] to gather component context and forward it to the binding engine. The binding engine generates an XML file containing the execution sequence and feedback mechanism. We call this part an analyzer and use it for autonomic self management in ubiquitous systems. The Traffic Manager manages the traffic directed to the service gateway.

4 Application Scenario

The Traffic Manager receives SOAP (spell out acronym on first use) requests from many devices within a cluster and redirects them to all the other internal parts of SMF. The Acceptor thread of the Traffic Manager receives a SOAP request (service request) and then puts it into the Wait Queue. The Wait Queue contains the latest context of the gateway load. If the gateway is in a saturated state, the service request is handled by the self-aware sub module. The figure 2 is the pseudo code of self-aware sub-module in WorkerManager's Delay Time Algorithm. Let a service request (SR_1) arrives at the gateway. At first the SR_1 is checked if it contains the comebacktime stamp (for fair scheduling). If the comebacktime is 'fair' (that is the service request is returned after the instructed time), it is forwarded to the Wait Queue else it is accessed against the work load of the Worker Manager. The Acceptor is updated about the latest status of the Worker Manager. The Acceptor evaluates the intensity of current workload (how long it will take to free resources) and adds buff (buffer is the time to give some extra room to gateway) to the time. The aggregate time is assigned to SR_1 and the service request is discarded.

When the system is ready to accept the service request, the Traffic Manager gets a Worker Thread from a thread pool and run it. The Worker Thread gets the delay time and the over speed from the WorkerManager. The admission to other internal parts SMF is controlled by the Worker Thread that accepts the arriving requests only if the over speed $OS(t_{i+1})$ at the time t_{i+1} is below zero and the delay time $D(t_i)$ at the time t_i is below the baseline delay δ. Otherwise the requests have to sleep for the delay time calculated by the WorkerManager. After the Worker Thread sleeps for the delay time, the Worker Thread redirects the requests to the Root Cause Analyzer, the RCF Manager, and the Scheduler. Finally, the Worker Thread adds the number of processed transaction after finishing the related transaction. After sleeping during interval time, the WorkerManager gets the number of transactions processed by all Worker Threads and the maximum transaction processing speed configured by a system administrator. And then, the WorkerManager calculates the TPMS (Transaction per Milliseconds) by dividing the number of transactions by the maximum transaction processing speed and calculate the over speed $OS(t_{i+1})$ that means the difference of performance throughput at the time t_{i+1} between the TPMS and the maximum transaction processing speed during the configured interval time. If the value of the over speed is greater than zero, the system is considered as in an overload state. Accordingly, it is necessary to control the overload state. On the contrary, if the value of the over speed is zero or less than zero, it is not necessary to control the transaction processing speed. For controlling the overload state, this paper uses the delay time algorithm of the WorkerManager.

A new delay time $D(t_{i+1})$ at the time t_{i+1} is dependent on $D(t_i)$, which means the delay time at the time t_i. The $N(t_{i+1})$ means the number of active *Worker Threads* at the time $t_{i+1.}$ If the $D(t_i)$ is zero, $D(t_i)$ must be set one. If the $OS(t_{i+1})$ is below zero and the delay time $D(t_i)$ at the time t_i is greater than the baseline delay δ. On the contrary, if the $D(t_i)$ is below the baseline delay, $D(t_{i+1})$ is directly set zero. In other words, because the state of the system is under load, the delay time at the time t_{i+1} is not necessary. Accordingly, the *Worker Thread* can have admission to other internal parts *SMF*. The baseline delay is used for preventing repetitive generation of the over speed generated by suddenly dropping the next delay time in previous heavy load state. When the system state is continuously in a state of heavy load for a short period of time, it tends to regenerate the over speed to suddenly increment the delay time at the time t_i and then suddenly decrement the delay time zero at the time t_{i+1}. In other words, the baseline delay decides whether the next delay time is directly set zero or not.

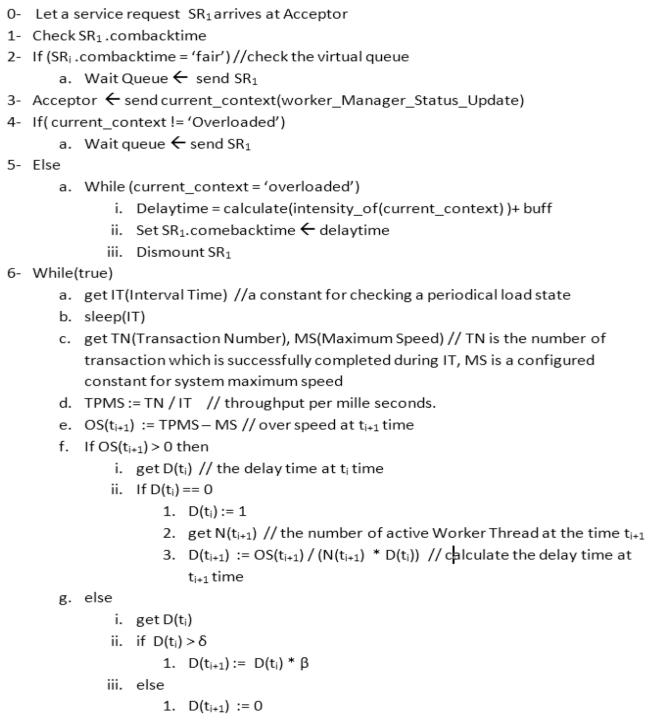
```
0-  Let a service request $SR_1$ arrives at Acceptor
1-  Check $SR_1$ .combacktime
2-  If ($SR_i$ .combacktime = 'fair') //check the virtual queue
        a.  Wait Queue $\leftarrow$ send $SR_1$
3-  Acceptor $\leftarrow$ send current_context(worker_Manager_Status_Update)
4-  If( current_context != 'Overloaded')
        a.  Wait queue $\leftarrow$ send $SR_1$
5-  Else
        a.  While (current_context = 'overloaded')
                i.  Delaytime = calculate(intensity_of(current_context) )+ buff
               ii.  Set $SR_1$.comebacktime $\leftarrow$ delaytime
              iii.  Dismount $SR_1$
6-  While(true)
        a.  get IT(Interval Time) //a constant for checking a periodical load state
        b.  sleep(IT)
        c.  get TN(Transaction Number), MS(Maximum Speed) // TN is the number of
            transaction which is successfully completed during IT, MS is a configured
            constant for system maximum speed
        d.  TPMS := TN / IT    // throughput per mille seconds.
        e.  $OS(t_{i+1})$ := TPMS – MS // over speed at $t_{i+1}$ time
        f.  If $OS(t_{i+1}) > 0$ then
                i.  get $D(t_i)$ // the delay time at $t_i$ time
               ii.  If $D(t_i) == 0$
                        1.  $D(t_i) := 1$
                        2.  get $N(t_{i+1})$ // the number of active Worker Thread at the time $t_{i+1}$
                        3.  $D(t_{i+1}) := OS(t_{i+1}) / (N(t_{i+1}) * D(t_i))$ // calculate the delay time at
                            $t_{i+1}$ time
        g.  else
                i.  get $D(t_i)$
               ii.  if $D(t_i) > \delta$
                        1.  $D(t_{i+1}) := D(t_i) * \beta$
              iii.  else
                        1.  $D(t_{i+1}) := 0$
```

Figure 2: The Pseudo Code for Self-Aware Module and WorkerManager's delay time Algorithm

The β percent of the Figure 2 decides the slope of a downward curve. However, if the delay time at the time t_i is lower than the baseline delay then the new delay time at the time t_{i+1} is set to zero. Accordingly, when a system state becomes heavy overloaded at the time t_i, the gradual decrement by β percent prevents the generation of repetitive over speed caused by abrupt decrement of the next delay time. Once the service request is received by the worker thread the analysis of the cause of anomaly starts. As proposed in [18] the faults can be single root cause based or multiple root cause based. We consider this scenario and classify a Root Cause Analyzer that checks the root failure cause through the algorithms proposed in [17]. After identifying the root causes, the Root Cause Fragmentation Manager (REFRCF Manager) looks up for the candidate plug-ins as solution. The RFCRCF manager also delegates the candidate plug-ins as possible replacement of the most appropriate. The scheduler schedules the service delivery mechanism as proposed in [18]. The processed fault signatures are stored in signature repository for future utilization. Let, N is concurrent service requests at the server at full time. This means that as soon as one thread finishes its execution, a new one will take its place. This assumption is made in order to insure that we analyze the worst case scenario of performance time with N service requests in execution queue. This means that the execution of a service request is done from its first until its last quantum (subparts of a service request i.e. analyze, evaluate, categorize etc) in the presence of other N-1 service requests.

- k→index variable spanning the service requests: 1<=k<=N
- S_j→CPU quantum length for server j ($s_{isolated}$ represents that value for a specific isolated server)
- I_{kj}→the number of cycles the kth service request needs to complete on server j.
- $i_{k\_isolated}$ represents that value for a specific isolated server

As soon as the execution queue is in a stable state, the time needed for the kth service request to complete in the presence of other N-1 service requests is

$$t_k = i_{kj} * s_j * N \quad (1)$$

The $i_{kj} * s_j$ this product represents in fact the execution time of the kth service request in isolation conditions (executed alone, without any other concurrent thread). This product $t_{k\ isolated} = i_{k\ isolated} * s_{isolated}$ is evaluated on an out of core server and is used as the base value for the load prediction. These two formulas are rather trivial and are standard results of queuing system. They mean that the prediction time for the execution depends on the total amount of connected service requests on the server.

Now a transaction is completed client request, and considering that the kth client permanently issues the same request to the server, then the number of transactions (Tx) that may be completed for k clients in interval T is

$$Tx_k = T / i_{kj} * s_j * N \quad (2)$$

from 1,

$$Tx_k = \Delta * T / t_{k\ isolated} * N \quad (3)$$

Where Δ is coefficient of CPU utilization. We can write 3 as,

$$Txcpu_K = \Delta * T / t_{k\ isolated} * N \quad (4)$$

From equation 4, it means that the execution time for a transaction depends upon the number of service requests in the active queue. So we can calculate the estimated time a CPU needs in order to get free from the requests in the active queue.

Let, there be N number of service requests present in the active queue. In time t_{k1}, the service request s_{k1} is being executed, the K_{N-1} service requests will reside in the memory.

- Ms_k→memory size kth service request. $Ms_k >= 1$
- B_k→branch statements in Ms_k in kth service request. $B_k >= 0$
- $Ttpb_k$→the Time needed per transaction.

$$Txmem_k = Ø * (((Ms_k * B_k) / Ttpb_k) * N) \quad (5)$$

Where Ø is coefficient of CPU utilization. Let, u(t) denote load of service request. We can normalize the service request as

$$y(t) = (u(t) - u_{min}(t)) / u_{max}(t) \quad (6)$$

Where $u_{min}(t)$ and $u_{max}(t)$ denotes the minimum and maximum load of the service request. According to equation 6, different service request traces and be compared with each other, while the impact of their internal analysis is eliminated. If we define T_k as the kth threshold for k=0,1,2…, k, then a function $y_{tk}=(t)$ is defined by

$$\text{iff } y(t) >= T_k \rightarrow y_{tk}(t) = 1 \text{ else} \rightarrow y_{tk}(t) = 0 \quad (7)$$

Generally $T_k = k*(1/k)$ Assuming $y_{Tk}(t) = 1$. From 6 and 7 we can say

$$y(t) <= (u(t) - u_{min}(t)) / u_{max}(t) \quad (8)$$

Or we can say,

$$Tnwbuff_k = \delta * \{(u(t) - u_{min}(t)) / (u_{max}(t))\} \quad (9)$$

where δ is coefficient of CPU utilization.

So combining 4, 5, 9 we get,

$$T_{comeback} = Txcup_k + Txmem_k + Tnwbuff_k \quad (10)$$

$$T_{comeback} = \{\Delta * T / t_{k\ isolated} * N\} + \{Ø * (((Ms_k * B_k) / Ttpb_k) * N)\} + \{\delta * ((u(t) - u_{min}(t)) / (u_{max}(t)))\} \quad (11)$$

Now we know that the service request has exponential distribution and the arrival rate has poison distribution. So we can say that the RTD (Round Trip Delay)

$$RTD = T_{comeback} + \text{queuing delay} + \eta_f + \eta_s \quad (12)$$

Where η_f is the propagation delay from server to the bottleneck link buffer that is the gateway buffer, and η_s is *the propagation delay* over the return path from the bottleneck link buffer to the client (the service request generator).

$$\mathrm{RTD} = \{\Delta * T / t_{k\,\mathrm{isolated}} * N\} +$$
$$\{\varnothing * (((Ms_k * B_k) / Ttpb_k) * N)\} +$$
$$\{\delta * ((u(t) - u_{min}(t)) / (u_{max}(t)))\} +$$
$$\text{queuing delay} + \eta_f + \eta_s \quad (13)$$

And efficiency is calculated as

$$E(S * N) = (N * T_s) / (S * 1 * T(S * 1))$$
$$\approx (N * T_s) / ((S)\, S * T) \quad (13)$$

In this expression, the range for S is $\left[1..\frac{N}{n}\right]$ and the range for N is $\left[1..k * \log_n N\right]$ which is less then $\left[2\log_n N\right]^n$ for HYWINMARC [3] based, and $\left[2\log_n \frac{N}{2} + N^{n^2}\right]^n$ for RoSeS [13] based solutions.

5 Simulation Results

In order to prove performance stability of the self-aware PLC-based autonomic self-healing system, we simulated the self-aware delay time algorithm of the WorkerManager. The maximum speed, δ and β for delay time algorithm are configured 388, 100ms, and 0.75 respectively. The figure 3.a shows that the gateway with PLC scheme is more stable than the one without PLC mechanism. The standard deviation at the gateway without PLC is more than 58.23 whereas the deviation in performance cost at the gateway with a PLC mechanism is 24.02 which proves the argument posted in the previous section that a PLC mechanism provides stability to gateways in u-zone based networks. The figure 3.b shows that applying self-aware sub module to the PLC mechanism gives stable performance than applying PLC algorithm only. The stability in the cost function with time shows that the cost is predictable over time scale. Although the cost of applying PLC mechanisms with a self-aware module is more than without it, the self-aware PLC gives more stability hence is more suitable in unpredictable, dynamic, and highly heterogeneous u-Zone Networks. The figure 3.c shows the gateway CPU performance. Whenever the gateway probability rises, it is reduced by the algorithm proposed in this paper whereas the gateway without self-aware algorithm goes into a crash state. We also get the results obtained from different experiments with varying parameters. We observe that increases in users increases the throughput. Increasing the β increases the capacity of the gateway to entertain a denser flux of service requests and hence increases the throughout. According to our experiments, the most important factor is the Maximum Speed configuration of the WorkerManager, while α and β are not directly related to the efficiency of the WorkerManager. From the experiments, we also check CPU usage and estimation of throughput when the proposed algorithm is in use and when it is not in use. The result is that when the maximum speed of WorkerManager is low, the throughput is low and vice versa. And there is also marked difference between the maximum throughput with no sleep times and throughout with different parameters and in the presence of sleep times. The proposed scheme gets 12% improvement in the existing scheme of no sleep-based date dissemination techniques.

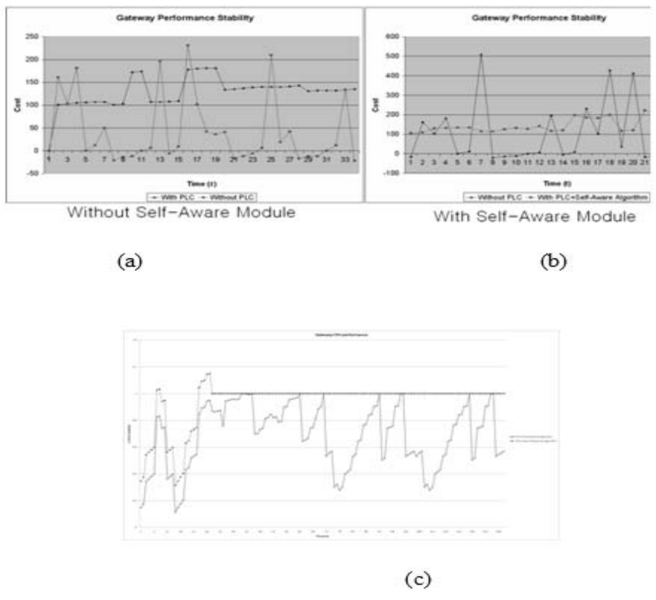

Figure 3: The Simulation Results for gateway performance stability

6 Conclusion and Future Work

In this paper we identify the role of self-healing which is mostly misunderstood among the modern day systems. This misunderstanding creates many logical problems especially in fault mapping, functional classification and categorization. We propose an Autonomic Healing-based Self-management Engine (AHSEN) that tackles the problem through a '*divide and conquer*' paradigm. We split the problem into smaller segments for coping with the wide variety of problems in a hybrid network. The self-management networks carry exponential complexity in their functionality. We propose to employ a self-aware load control scheme to manage the functionality of AHSEN. The simulation results show encouraging results and comparison with modern day solutions shows the significance of our work. In future we aim to study the effect of trust based interaction in the self-healing systems along with the signature independent anomaly identification at *NFM* level. We also aim to use the PLC mechanism to route the service requests launched in the sgateway's saturated state to the peer gateways so that the load on the host gateway can be distributed to the peer gateways.

Acknowledgments

This work was supported in part by TRUST (Team for Research in Ubiquitous Secure Technology), which receives support from the National Science Foundation (NSF award number CCF-0424422), and NSF S&T Center.

References

[1] Firetide www.firetide.com.

[2] Doufexi, A. Tameh, E. Nix, A. Armour, S. Molina, A. "Hotspot wireless LANs to enhance the performance of 3G and beyond cellular networks", Communications Magazine, IEEE, Publication Date: July 2003, Volume: 41, Issue: 7, On page(s): 58- 65 .

[3] Shafique Ahmad Chaudhry, Ali Hammad Akbar, Ki-Hyung Kim, Suk-Kyo Hong, Won-Sik Yoon," HYWINMARC: An Autonomic Management Architecture for Hybrid Wireless Networks" Network Centric Ubiquitous Systems (NCUS 2006).

[4] Burke Richard, 2004, "Network Management. Concepts and Practice: A Hands-on Approach", Pearson Education, Inc.

[5] Minseok Oh. Network management agent allocation scheme in mesh networks Communications Letters, IEEE Volume 7, Issue 12, Dec 2003 Page(s):601 – 603

[6] Kishi Y. Tabata, K.; Kitahara, T.; Imagawa, Y.; Idoue, A.; Nomoto, S.; Implementation of the integrated network and link control functions for multi-hop mesh networks in broadband fixed wireless access systems Radio and Wireless Conference, 2004 IEEE 19-22 Sept. 2004 Page(s):43 - 46

[7] S. Yong-Lin, G. DeYuan, P. Jin, S. PuBing, A mobile agent and policy-based network management architecture, Proceedings, Fifth International Conference on Computational Intelligence and Multimedia Applications ICCIMA 2003, 27-30 Sept. 2003, Page(s):177-181.

[8] Robert Steinke, Micah Clark, Elihu Mcmahon, "A new pattern for flexible worker threads with in-place consumption message queues", Volume 39 , Issue 2 (April 2005) Pages: 71 - 73 Year of Publication: 2005.

[9] Junaid Ahsenali Chaudhry, and Seung-Kyu Park, Some Enabling Technologies for Ubiquitous Systems, Journal of computer Science 2 (8): 627-633, 2006.

[10] Junaid Ahsenali Chaudhry, and Seungkyu Park, "Using Artificial Immune Systems for Self-healing in Hybrid Networks", in Encyclopedia of Multimedia Technology and Networking, Published by Idea Group Inc., 2006.

[11] Shelton, C. & Koopman, P., "Improving System Dependability with Alternative Functionality," DSN04, June 2004.

[12] Morikawa, H. (2004). The design and implementation of context-aware services. Proceedings of IEEE saint-w 2004, 293 – 298.

[13] D. C. Schmidt, "Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching," in Pattern Languages of Program Design (J. O. Coplien and D. C. Schmidt, eds.), pp. 529–545, Reading, MA: Addison-Wesley, 1995

[14] D. C. Schmidt, "Acceptor and Connector: Design Patterns for Initializing Communication Services," in Pattern Languages of Program Design (R. Martin, F. Buschmann, and D. Riehle, eds.), Reading, MA: Addison-Wesley, 1997.

[15] Junaid Ahsenali Chaudhry, Seungkyu Park, "A Novel Autonomic Rapid Application Composition Scheme for Ubiquitous Systems", The 3rd International Conference on Autonomic and Trusted Computing (ATC-06), 2006

[16]Wolfgang Trumler, Jan Petzold, Faruk Bagci, Theo Ungerer, AMUN – Autonomic Middleware for Ubiquitious eNvironments Applied to the Smart Doorplate Project, International Conference on Autonomic Computing (ICAC-04), New York, NY, May 17-18, 2004.

[17] Gao, J.; Kar, G.; Kermani, P.; Approaches to building self-healing systems using dependency analysis, Network Operations and Management Symposium, 2004. NOMS 2004. IEEE/IFIP Volume 1, 19-23 April 2004 Page(s):119 - 132 Vol.1

[18] Junaid Chaudhry, and Seungkyu Park, "On Seamless Service Delivery", The 2nd International Conference on Natural Computation (ICNC'06) and the 3rd International Conference on Fuzzy Systems and Knowledge Discovery (FSKD'06) 2006.

[19] Ilgun, K.; Kemmerer, R.A.; Porras, P.A., "State transition analysis: a rule-based intrusion detection approach," Software Engineering, IEEE Transactions on , vol.21, no.3pp.181-199, Mar 1995.

[20] T. F. Lunt, "Real-time intrusion detection," in Proc. COMPCON, San Francisco, CA, Feb. 1989.

[21] T. F. Lunt et al., "A real-time intrusion detection expert system," SRI CSL Tech. Rep. SRI-CSL-90-05, June 1990.

[22] J. Hu, I. Pyarali, and D. C. Schmidt, "Applying the Proactor Pattern to High-Performance Web Servers," in Proceedings of the 10th International Conference on Parallel and Distributed Computing and Systems, IASTED, Oct. 1998.

[23] P. J. Denning: Thrashing: Its Causes and Prevention. Proc. AFlPS FJCC 33, 1968, pp, 915-922

[24] Turing, Alan M., On Computable Numbers, with an Application to the Entscheidungs Problem. Proceedings of the London Mathematical Society, 2 (42):230-265, 1936.