A Novel Autonomous Management Distributed System for Cloud Computing Environments

Razvan-Ioan Dinita, George Wilson, Adrian Winckles, Marcian Cirstea, Tim Rowsell
Computing and Technology
Anglia Ruskin University
Cambridge, United Kingdom
{razvan.dinita, george.wilson, adrian.winckles, marcian.cirstea, tim.rowsell}@anglia.ac.uk

***Abstract*—This paper describes a novel modular design of an autonomous management distributed system (AMDS) for cloud computing environments and it presents its implementation with the Scala programming language. The AMDS was designed from the ground up with distributed deployment, modularity and security in mind, using a full object oriented approach. A key feature of this system is the ability to gather and store information from various networking and monitoring devices from within the same computing cluster. Another key feature is the ability to intelligently control VMWare vSphere local instances based on analysis of collected data and predefined parameters. vSphere in turn, once it receives commands from the AMDS, proceeds to issue instructions to multiple locally monitored ESXi severs in order to maximize energy efficiency, reduce the carbon footprint and minimize running costs. The predefined parameters are based on results from a previous paper written by the authors. The AMDS has been deployed on the authors' test bed and is currently running successfully. Test results show highly potential industrial applications in datacenter energy management and lowering of operating costs.**

Keywords—cloud; distributed; energy; optimisation; software

I. INTRODUCTION

Cloud computing is an emerging technology that devolves computing resources to the Internet [1]. The term "cloud" is commonly used to describe a cluster of computing hardware linked through a series of networking devices. Each computing component has a Hypervisor installed on it. The Hypervisor has been classified into two categories: type 1, which runs directly on the host's hardware, and type 2, which runs within a conventional operating system [2].

This paper focuses on presenting a novel design of an autonomous management distributed system. It continues the initial conceptual work done by the authors [3] and makes use of type 1 Hypervisors, which are capable of running multiple virtualized machines (VM) simultaneously, all controlled by an independent software package. VMWare, through their Academic Program, has supplied the solution currently deployed on the authors' test bed. The package is comprised of two main components: ESXi, a type 1 Hypervisor, and the vSphere Client, a complete cluster management solution.

Also, as presented by [4] energy efficient management of cloud infrastructures is still a very active and current issue in research, more so in the industry where better energy management usually brings lower datacenter operating costs. The authors have already covered this aspect in an earlier paper [3].

The authors have considered two approaches to optimising the energy management within the VMWare cluster. The first one looked at using the features provided by the software solution to set different parameters to achieve the desired results. Unfortunately, there were no relevant parameters available to be set within the vSphere client. The second approach, and the one presented in this paper, was to design a custom distributed software solution to interface with the vSphere client.

The use of distributed systems has its roots in operating system architectures that were initially studied in the 1960s [5]. The first widespread distributed systems were invented in the 1970s [6] and implemented on an Ethernet infrastructure. [7] also presents a well designed modular system of a SOA based architecture (also invented in the 1960s) for use in Cloud Computing environments. The authors have based their software solution design on the above presented architectures due to their tried and tested reliability for over 40 years.

II. DESIGN OF THE AMDS

For the benefit of the reader Fig. 1 presents an overview of the AMDS's position within a Cloud Computing environment. The AMDS connects to the four most important components of the cloud system: access point (connection to the outside world), power reading hardware (monitors power consumption), network reading hardware (monitors network flow), and the heart of the cloud system – the proprietary software (VMWare ESXi) that makes decisions regarding the server and storage management.

A. Design Goals

The AMDS was designed from the ground up with three main goals in mind: 1) security, 2) modularity, and 3) parallel processing.

To address security the authors have based their design on work done by [8] in order to introduce an appropriate authentication component. As such, each system component is required to authenticate each time it interacts with any other component.

Modularity has been achieved through implementing sound Object Oriented programming principles. Some core principles relevant to this work outlined by [9] have been applied to the design. Every task the system is required to accomplish is split down into independent, fully reusable modules.

The third goal, parallel processing, is achieved through the deployment of several instances of the AMDS across the test bed. The system has been designed to be deployed on a VM and, through its configuration parameters, locates other system instances in order to establish a live communication link. From thereon, the linked instances coordinate each other's tasks in order to ease the hardware load and maximise efficiency, making the AMDS highly scalable. The system's distributed design takes into account key points presented by the authors of [10].

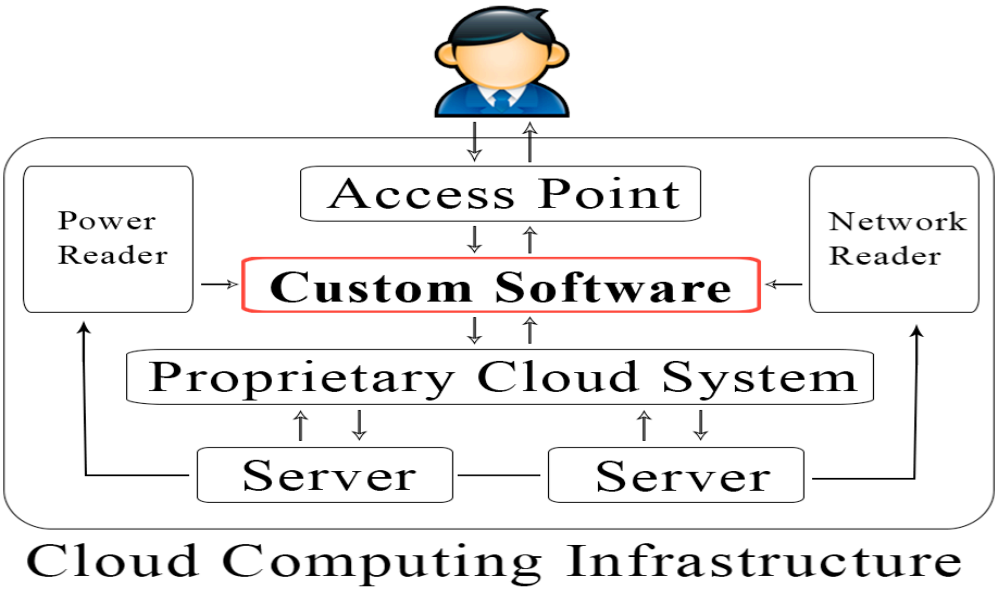

Fig. 1. Overview diagram of the AMDS's position within a Cloud Computing environment

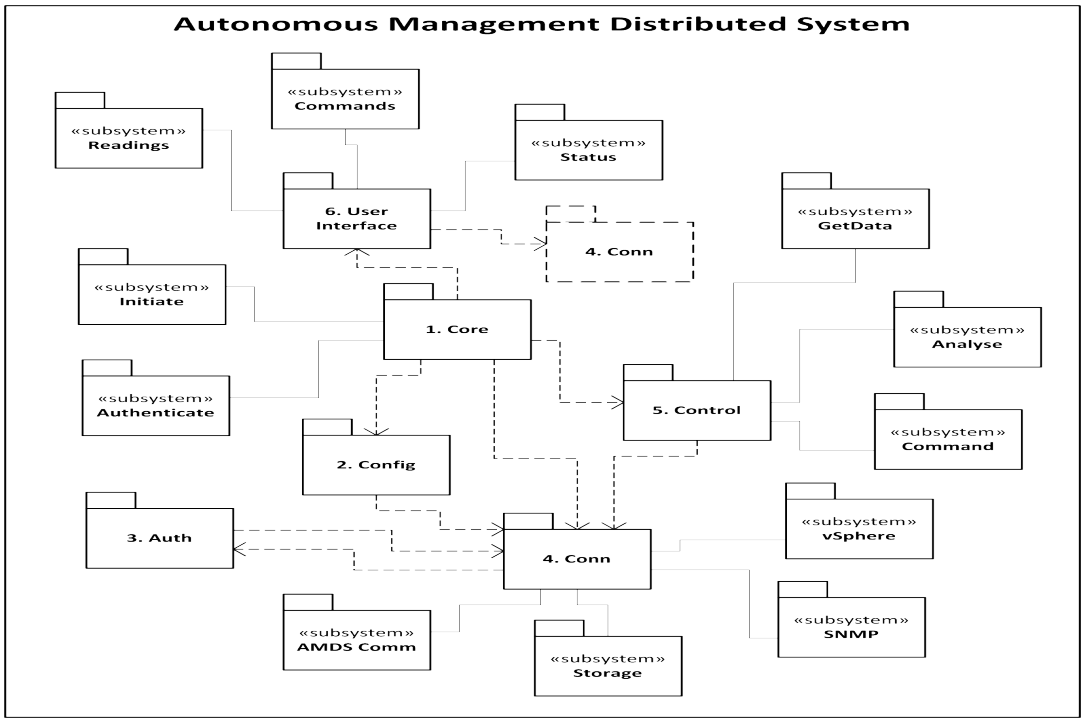

Fig. 2. UML Design Diagram of the AMD

B. Design Breakdown

Fig. 2 presents a UML [11] diagram of the AMDS design. At the time of writing it is composed of 19 different parts, each designed for a specific task and fully reusable. The main features of these component parts will now be described:

1) Core: Main system module. The entry point for the AMDS. From there the system accesses the configuration parameters (Fig. 2, 2. Config) and starts its internal tasks (Fig. 2, 5. Control) and the User Interface (Fig. 2, 6. User Interface). It is responsible for facilitating communication between the different modules attached to it.

a) Initiate: Module entry point. Achieves most of the module functionality. Initialises modules and establishes connections between them.

b) Authenticate: Helper module. Undertakes the initial system authentication. This helps with detection of possible hijack attempts by making sure a current instance remains valid and genuine.

2) Config: Helper module. Responsible for maintaining and providing access to the system configuration parameters. It interacts with the connection module (Fig. 2, *4. Conn*) in order to gain access to the Storage component (Fig. 2, *4. Conn :: Storage*), Config database. It is active throughout the lifespan of the system instance, facilitating on-the-fly parameter alteration.

3) Auth: Key system module. Manages task and connection authentication. It performs checks against the initial determined instance validity and genuineness in an attempt to discover potential system hijack attempts and prevent them. It locks down any connection or task that does not pass the validation step and makes a log entry with relevant details on the security issue.

4) Conn: Main system module. Facilitates all system connections between the modules themselves or between the modules and the computing cluster. It is the main access route to specialised mini-modules as well as attempt to authenticate each connection passing through it by calling upon the Auth module (Fig. 2, *3. Auth*).

a) AMDS Comm: Critical mini-module. Manages communication between instances of the AMDS including the passing of data between them. Acts as a load balancer by creating a bridge between current instance and one other instance. On each connection attempt it calls upon the Auth module to verify the integrity of the outside instance before allowing any kind of information exchange.

b) Storage: Main mini-module. Keeps track of internal databases for each system module that deals with data. It stores information for the Config and Control modules.

c) SNMP: Specialised mini-module. Facilitates passing of information between current system instance and devices that understand the Simple Networking Management Protocol (SNMP). This protocol allows data to pass both ways, making it possible to issue commands and receive results between different devices that use it.

d) vSphere: Critical specialised mini-module. Interfaces the VMWare vSphere client to allow issuing commands and retrieving results. This module bridges the gap between the custom designed AMDS and the proprietary software solution provided by VMWare.

5) Control: Main module. Initiates data collection, storing and analysis tasks, as well as initiate commands to the vSphere Client through the Conn module (Fig. 2, *4. Conn :: vSphere*). This allows for data to be collected from monitoring devices across the computing cluster, stored, analysed and actions to be taken based on the results and the configuration parameters.

a) GetData: Main mini-module. Deals with raw data retrieval. It initiates connections to the SNMP mini-module (Fig. 2, *4. Conn :: SNMP*)., retrieves and stores collected information using the Storage mini-module (Fig. 2, *4. Conn :: Storage*), Raw Data database.

b) Analyse: Main mini-module. Retrieves chunks of raw data from the Storage mini-module (Fig. 2, *4. Conn :: Storage*), Raw Data database, and come up with data capable of being compared to the configuration parameters. It then stores the analysis results using the same mini-module, but in the Results database.

c) Command: Main mini-module. Compares analysis results with the configuration parameters and make intelligent decisions which maximise energy efficiency. After it stores issued commands in the Commands database, it then proceeds to send them to the correct interfacing mini-module from within the Conn module (Fig. 2, *4. Conn*).

6) User Interface: Noncritical system module. Facilitates system monitoring by presenting information stored on the system in human readable form.

a) Status: Main mini-module. Provides an overview of the current system state as well as global statistics, including access to security logs and top level information on database disk usage.

b) Readings: Main mini-module. Provides an in-depth view of each individual database currently utilised by the system instance. All databases maintained by the Conn module (Fig. 2, *4. Conn*) are included. It makes use of data filtering and table display.

c) Commands: Main mini-module. Provides an in-depth view of all command decisions the urrent system instance has taken as well as the accompanying results received from all the commanded systems.

III. Implementation

A. Language Considerations

In the implementation stage the authors considered many programming languages capable of initiating remote

connections, including Ruby, PHP, Java, Scala, C++, C#. The main criteria to be considered was portability i.e. make the system so that it can be deployed on as many different operating systems as possible. Only two of the considered languages met the required criteria: Java and Scala.

Java is an established programming language making its first appearance in 1995[1]. It is able to function on any operating system running the Java Virtual Machine (JavaVM). All major systems, including Unix, Linux and Windows are currently capable of running the JavaVM. However whilst the language functionality continues to evolve it does not handle running multithreaded tasks well.

Scala [12] is a relatively new language. In spite of only making its first appearance in 2003, it has grown in popularity very quickly due to its multithreading capabilities as well as its concise way of expressing common programming patterns[2]. This has helped to drastically reduce development time on projects. One major advantage and the main reason why it has gained popularity so quickly is its seamless integration with Java. Scala support in Java for example can be provided by importing an appropriate library, and all Scala programs also run on the JavaVM which means that these can be deployed on all the major systems.

B. *Implementation Process*

Scala has a unique way of dealing with data structures – there are mutable (can be changed – e.g. *var*) and immutable (cannot be changed – e.g. *val*) variable types. Scala creators recommend using the immutable types because this minimises the risk of random or unintentional data corruption during runtime.

Scala also allows for almost out-of-the-box distributed code implementations through the use of Actors. Scala Actors are capable of independent and asynchronous operation, operating by passing messages from one to another. They function under a command hierarchy and also allow for quick error recovery. Since each actor operates independently of each other, if one encounters a fatal error, the message is cascaded up the chain of command until it reaches an actor programmed to handle that type of issue. It can then proceed to take further actions as necessary e.g. restart the failed actor.

In the development process the authors have used the Eclipse Integrated Development Environment (IDE) to assist with code completions and debugging as necessary. Each module has been implemented using inheritance based Object Oriented programming principles. Fig. 3 shows a few lines of Scala code (part of the Conn module implementation). Java libraries have been used to facilitate remote connections and Actors have been used to operate as message transporters. The code in Fig. 3 is set to receive remote connections and take different actions based on the type of result.

[1] http://www.oracle.com/technetwork/java/javase/overview/javahistory-index-198355.html
[2] http://www.scala-lang.org/node/25

NetworkModule.scala

```
package DNA

class ConnModule(cfg: Map[String, Map[String, Map[String, String]]]) {

    import akka.actor._
    import akka.util.ByteString
    import java.net.InetSocketAddress

    val util = new UtilitiesModule

    class Server(port: Int) extends Actor {

        override def preStart {
            try {
                IOManager(context.system) listen new InetSocketAddress(port)
                println("Started listening on port " + port)
            } catch {
                case e:Exception => println("Exception: " + e.getMessage)
            }
        }

        def receive = {
            case IO.Listening(server, address) =>
                println("The server is listening on socket " + address + ".")

            case IO.Connected(socket, address) =>
                println("Successfully connected to " + address)

            case IO.NewClient(server) =>
                println("New incoming connection on server from.")
                val socket = server.accept()
                socket write util.ToBytes(cfg("app")("info")("name") + " " + cf

            case IO.Read(socket, bytes) =>
                val sw = socket.asWritable
                val string = util.ToString(bytes)

                println("Received incoming data from socket: " + string + ".")
```

Fig. 3. Conn Module implementation, screenshot taken from Eclipse IDE

```
//3tier Map
def Get(): Config = {

    //imports
    import scala.xml.{XML, Utility}

    try {

        val nonTrimDoc = XML loadFile filePath
        val doc = Utility trim nonTrimDoc

        val returnList = new Config

        doc match {
            case <config>{topElems @ _*}</config> => //root
                for (topElem <- topElems) { //1st tier
                    topElem.nonEmptyChildren foreach( //2nd tier
                        elem => {
                            elem.nonEmptyChildren foreach( //3rd tier
                                innerElem => {
                                    returnList + (topElem.label, elem.label,
                                        innerElem.label, innerElem.text)
                                }
                            )
                        }
                    )
                }
            case elem @ _ => println("Error, unmatched element found: " +
                elem.toString)
        }

        returnList

    } catch {
        case e:Exception =>
            val exception = new Config
            exception + ("Error", "Info", "Message", e.getMessage)
            exception
    }
```

Fig. 4. Part of the Storage Module, Config database, screenshot taken from Eclipse IDE

The databases have been implemented using the eXtensible Markup Language (XML). Fig. 4 shows part of the Storage module, specifically part of the method that deals with parsing the Config database information. The code uses Scala libraries for dealing with XML data structures and builds up a Config type object (first line, after the colon, signifies the return type). Exception handling is implemented in every module as a safety measure to prevent unexpected program termination due to unexpected or corrupt data.

Another important component of the ASDM is the vSphere mini-module. In its development the authors have made heavy use of the vSphere SDK for Java [13]. This set of Application Programmable Interfaces (APIs) facilitates message exchanges between the Control module and the vSphere Clients currently in the computing cluster. Still in its preview stages, it allows for much interaction between third party programs and the vSphere client itself.

IV. Results

The ASDM is currently deployed and running on the authors' test bed. An Ubuntu Linux based Virtual Machine was chosen to run the application due to the high reliability of the Operating System. The VM has is connected to the ESXi Servers via a closed Virtual Network. This ensures seamless connectivity between virtual and physical hardware, thus allowing the AMDS to receive information from the ILOs and the vSphere Client and to send commands back.

At the moment, data is being collected and analysed for debugging purposes. As seen in Fig. 5, the program is running successfully. The core initializes and authenticates as expected; Conn, SNMP and Control modules are being activated, and raw data starts to be collected for storing and analysis.

```
Distributed Network Application v0.1 Alpha
Core/Run Initialising...
Initialising Config Module...
Done.
Authenticating...
Done.
Initialising Conn Module...
Done.
Initiating SNMP module...
Done.
Initiating Control Module...
Done.
Running GetData from Control Module...
Connection established...
Waiting for data...
Done
```

Fig. 5. Output from ASDM, console view

Since the AMDS has been deployed it has produced a great number of data stored within several databases. The data comes from queries performed by the AMDS on the different networking hardware operating within the cloud environment (Switches, Routers, ILOs, ESXi). The authors have merged and analysed all generated data, the results of which have been expressed in Table I and Fig. 6.

The system efficiency was calculated by using the formulas (1) and (2). In formula (1) P_{rise} is the power consumption rise percentage calculated by dividing $P_{current}$ (server power consumption at any other time – processor load > 0%) by P_{idle} (server power consumption when idle – 0% processor load). In formula (2) E (server operating efficiency) is calculated by dividing L (server load) by P_{rise}.

$$P_{rise} = P_{current} / P_{idle} \quad (1)$$

$$E = L / P_{rise} \quad (2)$$

TABLE I. Comparison between normal system operation (without AMDS) and optimized system operation (with AMDS)

	Before AMDS	During AMDS operation
Processor Load (%)	25	100
Power Consumption (Watts / Hour)	168	239
Power Consumption Rise (Watts / Hour)	35.5	92
Efficiency (%)	70	108

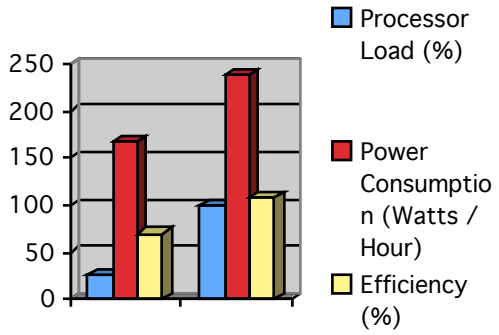

Fig. 6. Comparison between normal system operation (without AMDS) and optimized system operation (with AMDS)

In the second column of Table I the data has been recorded before the AMDS has been enabled (decision making module was disabled). The system efficiency stabilized at 70% due to the fact that several servers operating at 25% of their potential hardware load. The server power consumption at this stage was 168 Watts / Hour, an increase of 35.5% from system idle state.

In the third column of Table I the data has been recorded after the AMDS has been enabled (decision making module was enabled). Almost immediately all system traffic had been redirected towards one of the active servers while the others had been shut down to conserve power, thus bringing the system efficiency up to 108%. Power consumption at this stage was 239 Watts / Hour, an increase of 92% from system idle state.

The results presented above demonstrate how the AMDS is capable of minimising the cloud system power consumption

by up to 8%, thus generating an important operating cost reduction.

V. CONCLUSIONS

The development of ASDM is an ongoing process. The authors are constantly tweaking and changing the code in pursuit of better performance. The proposed system has several key industrial applications:

1) Green Datacetre. The proposed system is capable of reducing overall energy consumption by intelligently turning physical servers on and off based on data collection from throughout the computing cluster.

2) Lower Datacentre operating costs. This is a direct consequence of the previous statement. Overall lower energy consumption leads to reduced operating costs. This in turn allows for higher profits and more investments to be made.

3) Set-and-forget scenarios. The AMDS, due to its autonomous nature and modular design, is capable of on-the-fly self reconfiguration based on analysis results of gathered data. This flexibility makes it ideal for set-and-forget situations as well as low cost maintenance schedules.

Next, the authors have started looking at intrusion detection in an attempt to develop a successful method of hijack detection. Data collected from running ASDM, including the Databases and Logs, is collected at the same time as an attempt at hijacking the system is in progress. All data is continuously monitored and analysed for signs of the hijack process on the system.

REFERENCES

[1] Mirashe, S. P.; Kalyankar, N. V.; (2010), "Cloud Computing", *Communications of the ACM*, 51 (7), 9.

[2] Popek, G. J.; Goldberg, R. P.; (1974). "Formal Requirements for Virtualizable Third Generation Architectures", *Communications of the ACM,* 17 (7), pp. 412 –421, doi:10.1145/361011.361073.

[3] Dinita, R. I.; Wilson, G.; Winckles, A.; Cirstea, M.; Jones, A., "Hardware Loads and Power Consumption in Cloud Computing Environments", *ICIT 2013 – 2013 IEEE International Conference on Industrial Technology*, pp. 1291-1296, 25-27 Feb. 2013, ISBN: 978-1-4673-4568-2.

[4] Mora, D.; Taisch, M.; Colombo, A. W., "Towards an energy management system of systems: An industrial case study," *IECON 2012 - 38th Annual Conference on IEEE Industrial Electronics Society*, pp. 5811-5816, 25-28 Oct. 2012.

[5] Andrews, G. R. (2000), "Foundations of Multithreaded, Parallel, and Distributed Programming", p. 348, Published by Addison–Wesley, ISBN 0-201-35752-6.

[6] Andrews, G. R. (2000), "Foundations of Multithreaded, Parallel, and Distributed Programming", p. 32, Published by Addison–Wesley, ISBN 0-201-35752-6.

[7] Karnouskos, S.; Colombo, A.W.; Bangemann, T.; Manninen, K.; Camp, R.; Tilly, M.; Stluka, P.; Jammes, F.; Delsing, J.; Eliasson, J., "A SOA-based architecture for empowering future collaborative cloud-based industrial automation," *IECON 2012 - 38th Annual Conference on IEEE Industrial Electronics Society*, pp.5766-5772, 25-28 Oct. 2012.

[8] Irani, G.N.H.; Tawosi, V., "AAMA: A new Authentication and Authorization architecture for modular information systems, a robust object oriented approach," *Application of Information and Communication Technologies (AICT), 2011 5th International Conference on*, pp. 1-5, 12-14 Oct. 2011.

[9] Caragiozidis, M.; Mouratidis, N.; Kavadias, C.; Loupis, M.; Berger, M., "Design Methodology for a Modular Component Based Software Architecture," *Computer Software and Applications, 2008. COMPSAC '08. 32nd Annual IEEE International*, pp. 1122-1127, July 28 2008 - Aug. 1 2008.

[10] Maffei, A.; Hofmann, A., "From flexibility to true Evolvability: An introduction to the basic requirements," *Industrial Electronics (ISIE), 2010 IEEE International Symposium on* , vol., no., pp.2658,2663, 4-7 July 2010.

[11] Unified Modeling Language, http://www.uml.org.

[12] Scala Programming Language, http://www.scala-lang.org.

[13] VMware vSphere™ SDK for Java, http://communities.vmware.com/community/vmtn/developer/forums/java_toolkit