Personal Autonomic Computing Self-Healing Tool

Roy Sterritt Saulai Chung
School of Computing and Mathematics,
Faculty of Engineering
University of Ulster
Northern Ireland
r.sterritt@ulster.ac.uk

Abstract

The objective of the research reported in this paper was to develop a proof of concept self-healing tool for the personal computing environment operating in a peer-to-peer mode and consisting of a pulse monitor and a vital signs health monitor. The prototype developed in Java, JNI and C utilising UDP to communicate to its peers proved the feasibility of the pulse and vital signs concepts and their ability to provide some self-healing properties within a PC environment. The functionality also opens new opportunities to provide self-configuring, self-optimising and self-protecting autonomic capabilities to personal computing.

1. Introduction

Self-healing, an emerging research discipline [1] is considered one of the four autonomic computing [2] properties required to achieve self-managing systems [3]. This is often harder to obtain for personal computing due to its flexible nature and diverse user base [4].

The pulse monitor has been recommended as an extension of the Globus Heartbeat Monitor (HBM) for Grid computing [5], as a construct within an autonomic manager [6][7] and a reflex mechanism within a telecommunications fault management architecture [8]. This paper looks at utilising the pulse monitor together with a health check mechanism in a PC environment to construct a self-healing tool. The tool operates in a peer-to-peer (P2P) mode without any additional environment on top of the Windows OS.

Section 2 discusses the background technologies; AC, Personal AC, P2P, heartbeat and beacon monitoring. Section 3 looks at the tool's design followed by its implementation in section 4. Finally section 5 concludes and summarises the main points.

2. Background

2.1 Autonomic Computing (AC)

IBM introduced the autonomic computing initiative in 2001, with the aim to develop self-managing systems [9]. With the growth of the computer industry, with notable examples being highly efficient networking hardware and powerful CPUs, autonomic computing is an evolution to cope with rapidly growing complexity of integrating, managing, and operating computing based systems. Computing systems should be effective [7], they should serve a useful purpose when they are first launched and continue to be useful as conditions change. The realization of autonomic computing will result in a significant improvement in system management efficiency. The disparate technologies that manage the environment work together to deliver best performance results [2].

The autonomic concept is inspired by the human body's autonomic nervous system [2]. The autonomic nervous system monitors heartbeat, checks blood sugar levels and keeps the body temperature normal without any conscious effort from the human. There is an important distinction between autonomic activity in the human body and autonomic responses in computer systems. Many of the decisions made by autonomic elements in the body are involuntary, whereas autonomic elements in computer systems make decisions based on tasks chosen to delegate to the technology [2].

Upon launching Autonomic Computing IBM defined four key self properties; self-configuring, self-healing, self-optimizing and self-protecting [3][2]. In the few years since the self-x list has grown as research expands bringing about the general term *selfware*, yet these four initial self-managing properties along with the four enabling properties; self-aware, environment aware, self-monitor and self-adjust, cover the general goal of self management.

2.2 Personal Autonomic Computing

Personal Autonomic Computing is AC in a personal computing environment [4]. In some respects, achieving autonomic computing within server environments will be an easier task than within personal computing. Servers are likely to have received the level of investment to ensure in-built fault tolerance and include extensive redundancy – including facilities such as 'hot swapping' [10]. Personal devices are often machines built on the *faster, cheaper and smaller* philosophy with limited, if any redundancy. Servers are also likely to have a user base of highly skilled teams, whereas personal devices are often in the hands of non-technical users who often also act as the administrator. Other considerations are required for personal computing such as flexibility of location (e.g. laptops) and of hardware (e.g. palm devices) and software configuration that complicate further the goal of achieving autonomic computing [4][7].

Examples of autonomic capabilities within personal computing are;

- *Self-configuring* Microsoft Windows XP has an automatic update function. It updates itself to catch updated or newly released component(s) [4].
- *Self-healing* Windows XP Professional provides a checkpoint function to backup the system and recover up to the checked point if the system has crashed.
- *Self-optimizing* Microsoft Windows XP Professional now optimizes the user interface based on the way the system is used. For instance it attempts to keep the desktop clean and uncluttered by removing items not recently used. Due to the nature of personal computing the user is asked to confirm that these changes take place [4].
- *Self-protecting* An example of a protection mechanism is encryption. Windows XP is built with an encryption capability that allows directories to be encrypted. Microsoft Internet Explorer is embedded with security protocols such as SSL and TSL. Norton's Antivirus (Symantec Corporation) software automatically scans all emails to check if they contain any virus. Microsoft Excel prompts an alert if the user opens a spreadsheet containing a macro which may have a virus.

2.3 Peer-to-Peer (P2P)

Peer-to-Peer (P2P) is a paradigm in which each workstation on a network has equivalent capabilities and responsibilities [11]. This differs from the Client/Server architecture, in which a Server is a dedicated computer to serve Client requests. The Server machine is usually always available so that Clients can connect to it at anytime. Peer-to-Peer is not a new concept; IP routing is peer-to-peer. To make P2P distinctive, nodes must operate outside the DNS (Domain Name Server) system and each node has significant autonomy from central servers. P2P computing offers a company a cost-efficient way of sharing computer resources, improving network performance, and increasing overall productivity.

In traditional P2P networking, computers are connected together as a workgroup and configured for the sharing of resources such as files and printers. In particular, the computers are located near each other physically and run on the same networking protocols. Today, computers are connected together over the Internet. Computers (including hand-held devices) can join the network from anywhere with little effort.

Peer-to-Peer architectures enable computers to share services and resources directly between one another. Computers range from a large server to a handheld device. Resources and services include the exchange of information, processing cycles, cache storage, and disk storage. P2P technologies benefit distributed computing as it provides efficient communication and quality of service [12]. The function of one of the P2P technologies is for reclaiming unused computing cycles on desktop computers and harnessing them into a virtual supercomputer [13]. In this platform, a large job can be broken into small pieces and run on separate machines in parallel. At the same time, it reduces the load on servers hence allowing them to perform specialized services more effectively. In the P2P-enabled distributed computing model, a managing server is configured to send different pieces of one computing job to a set of peers, who then distribute it on to 2nd-tier peers, then 3rd-tier peers, and so on. Collaboration in P2P computing is allowing teams which are in different geographic areas to work together. As with file sharing, collaboration can decrease network traffic by eliminating e-mail and decreases server storage needs by storing files locally, the result increases productivity. P2P computing also allows networks to work together using intelligent agents. Agents work on a workstation and communicate various kinds of information back and forth [12]. Agents may also initiate tasks on behalf of other systems on different workstations. A virus alert is an example.

In this research the peers form a 'neighbour-hood watch' scheme—looking out for each others health.

2.3 Heartbeat and Beacon Monitoring

Within Grid Computing, the OGSA (Open Grid Services Architecture) has a facility referred to as the Globus Heartbeat Monitor (HBM) which is designed to detect and report whether registered processes are still alive or not [14], by providing or failing to provide a 'heartbeat'. The heartbeat monitor may be considered a specific type of environment-awareness since from a system perspective these heartbeats provide awareness of the individual functioning elements [7].

The DS1 (Deep Space 1) [16][17] was launched in July 1998 by NASA. The beacon monitor was one of the twelve new feats of technology used in DS1. Its goal was to decreasing the total volume of down linked engineering telemetry, through reducing the frequency of downlink and the volume of data received per pass [17]. With beacon monitoring, the spacecraft assesses its own health and will transmit one of four sub-carrier frequency tones to inform the ground how urgent it was to track the spacecraft for telemetry [15]. Table 1 summarizes the tone definitions.

Table 1 – Beacon tone

Tone	Description
Nominal	All functions as expected No need to downlink
Interesting	Interesting – non-urgent event. Establish communications when convenient.
Important	Communications need to take place within timeframe or else state could deteriorate.
Urgent	Emergency. A critical component has failed. Cannot recover autonomously and intervention is necessary immediately.
No Tone	Beacon mode is not operating

The tones are generated by phase-modulating the RF carrier by a square-wave sub-carrier using 90 degrees modulation angle. The resulting downlink spectrum will consist of tones at odd multiples of the sub-carrier frequency above and below the carrier. Only the tones at the fundamental frequency will be used to represent the transmitted message.

The two primary flight software innovations implemented through the beacon monitor are onboard engineering data summarization and beacon tone selection [17]. The tone selector module maps fault protection messages to beacon tone states. Transforms and adaptive alarm thresholds are the components to create top-level summary statistics, episode data, low-resolution "snapshot" telemetry, and user-defined data. These two components aim to minimize the number of false alarms.

3. Self-Healing Tool Design

The assumption behind the tool is that dying/hanging processes on the PC are signs or indicators to the health of that PC. These *vital signs* may indicate that the PC is becoming unstable and possibly in immanent danger of hanging or unreliable for current processes running on that machine. As well as restarting the detected hung process(es) the peers are notified of the situation via a change in *pulse*.

This is particularly useful in situations where the PC is unattended for example running a web server, and the user may be notified via a peer PC that the machine is in difficulty. Another useful situation is when machines in the peer group are sharing work load, for example via Harmony PC grid services [18]; a peer is notified in advance of immanent danger and can recover data and re-allocate work to another peer. Such an approach is more proactive than responding once the machine has hung, and as such offers fuller potential for autonomic capabilities.

The underlying functionality of the tool is a heart-beat monitor; if a process hangs it should be restarted and the pulse monitor takes note. Upon several processes hanging or the same process repeatedly hanging within specified timeframes, a change occurs in the monitor's perception of how healthy the machine is and as such brings about a change in the pulse being broadcast from that PC.

Since the tool operates in a P2P mode it also takes responsibility to watch out for its neighbours; as such other PCs (peers) will register with it and it will monitor their pulse.

Figure 1 depicts an overview of the Pulse Monitor construct. An internal monitor inside a host takes care of monitoring its health condition which is represented by a Pulse. Each host is able to send its Pulse to a peer via an external monitor. The 'knowledge & database' stores the pulse level and rules (i.e. predefined knowledge) which may adapt over time; the monitoring logs; and the history of neighbour hosts. A computer system is different from a biological system; human biology reflection is involuntary while the decision making in computer systems is based on a set of predefined rules or policies. For example, rules such as the 'pulse sending interval' and 'terminate the failed process after three trials of re-starting the process', are re-configurable.

The host sends the degree of urgency to the peer's pulse external monitor instead of just a 'beat'. The urgency level is transformed based on the number of failed processes (Table 2).

The amount of processes required to cause a change in pulse is adaptable and need not necessarily remain at the values depicted in Table 1, as is the time window for qualifying failing processes.

Similar to the connection between the Local Monitor and Data Collector of the Globus HBM, the connection between two hosts is established using the UDP (User Datagram Protocol). TCP (Transmission Control

Protocol) provides a reliable, connection-oriented, continuous-stream service. However, TCP requires a significant amount of overhead. In contrast to TCP, UDP has a low-overhead. As this tool only transmits small size messages, UDP is more suitable than TCP.

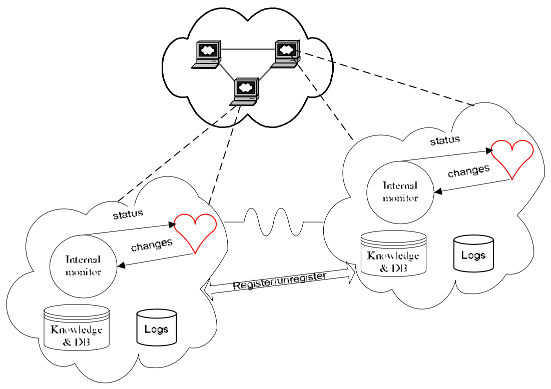

Figure 1 – Pulse Monitoring

Table 2 – Pulse value

Urgency Level	Description	Pulse Change Trigger (adaptable)
0	Nominal	no failed process
1	Interesting	1 failed process
2	Important	2 failed processes
3	Urgent	3 or more failed processes
—	No Pulse	Pulse monitor, or comms has failed

Figure 2 summarizes the functionality of the pulse monitor API. It scans the host periodically to check its health condition; it transforms the health condition to a pulse value and will send it to connecting neighbours (if any). If a process is found to have failed, the tool will try to re-start that process.

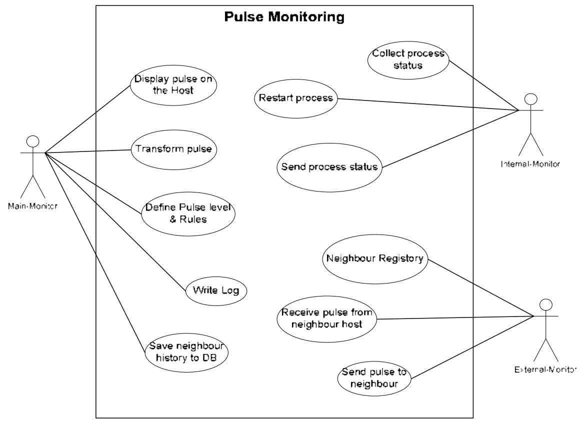

Figure 2 – Pulse Monitoring use-case diagram

4. Self-Healing Tool Implementation

This section looks at the implementation of the proof of concept.

4.1 Health Monitor Implementation

The Pulse Monitor is developed in Java. The Health-Monitor operates under the Microsoft Windows environment using the running processes as vital health signs. Since this health component is OS specific it is not developed in Java but C with the Windows SDK used to collect the process information.

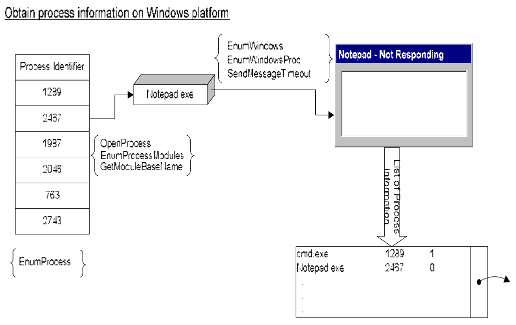

Figure 3 – Obtain process information

In the Windows environment, applications consist of executable files and DLLs [19]. A running application is known as a process. A process consists of one or more threads. A thread is the basic unit to which the operating system allocates processor time to execute its process code. Each process is assigned an identifier, and is valid until the process terminates. A module is an executable file or DLL. Each process consists of one or more modules [19].

Figure 3 illustrates how to obtain the list of process information in Windows platform. The performance monitoring in the Windows Platform Software Development Kit has the technologies to deal with process, thread, module, heap, processor, memory and event. The Process Status Helper (in psapi.dll) provides an interface to obtain information about processes [19]. The Windows system maintains a list of running processes. The EnumProcesses function retrieves all running process identifier. The OpenProcess function opens an existing process object to obtain the handle of a process. The EnumProcessModules function retrieves a handle for each module of a process. The GetModuleBaseName function retrieves the name of a module. A list of running processes with their identifier and name can be obtained by using these functions. However, the list does not have the processes status. To find out if the process is running normally, or if it has hung, it is necessary to first obtain the window of that process. Next, send a message to the window to see if it can respond or not. The EnumWindows function enumerates all top-level windows and as such has to be called with the EnumWindowsProc function. The EnumWindowsProc function is an application defined callback function. It receives top-level window handles. It is a placeholder for the application defined function. The window handle associated with a process is then passed to the SendMessageTimeout function to check if the window is responding or not. It returns without waiting for the time-out period to elapse if the window appears to not respond or has hung.

The Health-Monitor will terminate a process if the process has failed but can't be recovered (or re-started). The TerminateProcess function terminates a process and all of its threads. It stops execution of all threads within the process and requests cancellation of all pending I/O.

4.2 Health Monitor and Pulse Monitor Interfacing

Java Native Interface (JNI) [20] is used to interface between the Java based Pulse Monitor and the C coded Health Monitor. JNI defines a standard naming and calling convention so the Java Virtual Machine (JVM) can locate and invoke native methods. Within JNI, native methods can create, update, and inspect Java objects; Java can pass any primitive data types or objects as parameters to native methods; native methods can return primitive data types or objects back to the Java environment; Java instance or class methods can be called from within native methods; native methods can catch and throw Java exceptions.

The interface could have been developed in COM or J/Direct instead of the JNI approach, however these provide solutions which are even more OS specific [21]. With the approach used, in order to make the Health-Monitor run on different platforms such as UNIX, simply modify the collect process information methods in the C program.

Since the process list (the source of health indicators) is dynamic, an array is not flexible enough to store the list. A Vector class is used to hold the process information. The Vector class in Java is designed to store heterogeneous collections of objects thus providing methods for working with dynamic arrays of varied element types. Three Vector variables are declared in the Java program to hold process information; process name, process identifier, and process status. A function is defined in the Java program for the C program to add elements into these Vector variables. The FindClass function returns a reference to a class. The GetMethodID function performs a symbolic lookup on a given class and returns the method ID of an instance method. The CallObjectMethod is the function to invoke the method call of the found instance method.

4.3 Pulse Monitor Implementation

The External-Monitor provides the communications function with other hosts, using UDP (User Datagram Protocol) sockets (see Figure 4). UDP is described as unreliable, connectionless, and message-oriented [22] yet is good for sending short messages like those required for the Pulse Monitoring application, where all messages are less than 100 bytes. A socket is a handle for a communications link over the network to another application [22]. Sockets are often used in client/server applications whereby a centralized service waits for remote machines to request resources, handling each request as it arrives. For clients to know how to communicate with the server, it must know the port number on which the server is waiting. A client must then bind to this port to establish a socket connection. Two applications cannot bind to the same port on the same machine simultaneously. In TCP/IP protocol, ports used for standard services are well-known [23], for instance port numbers below 1024 are reserved and cannot be used. The well-known ports are controlled and assigned by the IANA (Internet Assigned Numbers Authority) and on most systems can only be used by system processes or by programs executed by privileged users. For example, port 21 is for FTP service; port 23 is for Telnet service; port 25 is for SMTP service and port 80 is for HTTP service.

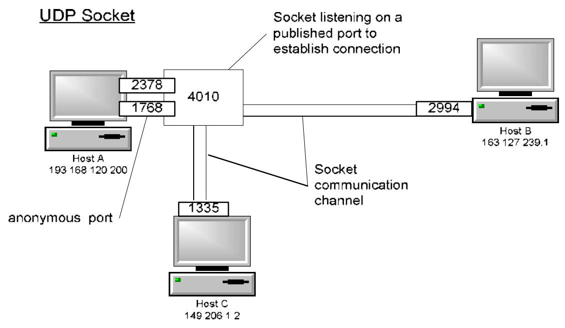

Figure 4 – Socket communication

However, there are other options available for sending messages between remote machines. The Remote Method Invocation (RMI) provided by Java has the technology for distributed systems. The RMI feature enables a program running on a client computer to make method calls on an object located on a remote server machine [20]. The RMI concept is based on an object advertising itself to the world by registering with a naming service (RMI Registry) and a client binds to an object through this RMI Registry. This is not an appropriate choice on the Pulse Monitoring application for two reasons: 1. the application is in peer-to-peer mode not client/server mode; 2. it is not ideal for a host that always has to advertise itself to the world.

In Java UDP programming, UDP communication carries out the following: creates an appropriately addressed datagram to send; sets up a socket to send and receive datagrams for a particular application; inserts datagrams into a socket for transmission; waits to receive datagrams from a socket and decodes received datagrams to extract the message, its recipient, and other meta information. The DatagramSocket class provides a function to create socket object and packets communication [20]. A datagram socket is the sending or receiving point for a packet delivery service. Each packet sent or received on a datagram socket is individually addressed and routed. The DatagramPacket class represents a datagram packet, which are used to implement a connectionless packet delivery service. Each message is routed from one machine to another based solely on information contained within that packet. A packet is a self-contained message that includes information about the sender, and length of the message, and the message itself. The send function sends out a datagram packet to a destination address. The receive function blocks until a datagram packet is received. It waits for a packet forever unless a timeout is enabled.

Before two hosts can send the pulse to each other, they first have to register to each other. When the External-Monitor starts, it immediately connects to its registered neighbour. The External-Monitor disconnects from all connecting neighbours when it ends. Un-registering from a neighbour will remove that host from its neighbour list and they no longer send the pulse to each other.

As mentioned above, unlike TCP, UDP is an unreliable service protocol. The use of only one port to serve all messages may overload a port and hence increase the probability of loosing a message. There are six UDP sockets created on different ports to wait for incoming messages;

- to register to it, port 4001
- to un-register from it, port 4002
- waiting neighbours connecting to it, 4003
- waiting neighbours disconnecting from it, 4004
- neighbours sending pulse to it, each host defines its own port number
- waiting neighbours to check if the host is still on or not, 2222

Timeout is not enabled on these sockets because they have to wait for incoming messages forever. When it receives a message, it then calls the corresponding function and replies an acknowledgement to the sender. The reason for dedicating different ports to serve a particular purpose is to minimize the loss of messages. To send a message, a separate socket port is open. The timeout is enabled to wait for a reply to ensure the message is delivered. When finished, this socket will close.

As stated the health tool and pulse monitor are made up of four components; Main-Monitor, Internal-Monitor, External-Monitor, and Health-Monitor (Figure 5): All components have to be executed synchronously, since multiple jobs are required to be carried out at the same time. Java has built-in support for threads through which it is possible to achieve multitasking. The Thread class implements Runnable interface by default. The Runnable interface enables a class to execute code in its own thread. The Runnable interface has an abstract function run. The code of these monitors is written in the run function. The job will only start when the start function is called.

The sleep function causes a thread to pause for a dedicated period of time. The Internal-Monitor sends process status to the Main-Monitor periodically. After it sends all process status to the Main-Monitor, it will sleep (for a defined period) to give the CPU time to handle other components. The Main-Monitor does the same, when it receives process status, it will carry out the appropriate action and when complete it will sleep for a predefined period of time. The yield function works like sleep, allowing other threads to execute and when it is complete, the CPU time is returned to the current thread object. The destroy function destroys a thread object without any cleanup.

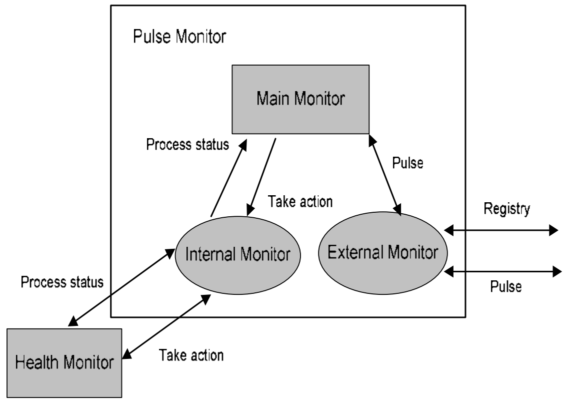

Figure 5 – Pulse Monitor

With reference to Figure 5; the Main-Monitor is a coordinator in the Pulse monitoring component. It initiates the Internal-Monitor and the External-Monitor, whereby the Main-Monitor communicates with the Internal-Monitor which in turn sends process status to the Main-Monitor which determines what action to take based on each process status. The Main-Monitor would then send the decision to the Internal-Monitor. Concurrently, the Main-Monitor communicates with the External-Monitor, when it receives process status from Internal-Monitor, transforming the process status into a Pulse. The Pulse is then sent to the External-Monitor to propagate to its neighbours. When the External-Monitor receives a Pulse from its neighbours, it sends the pulse to the Main-Monitor which outputs the received neighbour status into a log file.

The Internal-Monitor is responsible for the monitoring of processes running on the machine. The Health-Monitor is responsible for obtaining process information. The External-Monitor is responsible for the communication between its neighbours.

5. Summary and Conclusion

Today, computer-based systems are more and more difficult to manage. Autonomic computing helps to address the complexity issues by using technology to manage technology. A computer system is self-managing if it has some self-configuring, self-healing, self-protecting, and self-optimizing properties [3][2]. Self-healing is concerned with ensuring effective recovery when a fault occurs without human interaction. To achieve the self-healing objective, a system must be self-aware and environment-aware. In a biological system, the human body reacts with the external environment involuntarily; while in computer-based systems, autonomic elements make decisions based on the available technologies.

The objective of this research was to develop a proof of concept self-healing tool for the personal computing environment operating in a peer-to-peer mode consisting of pulse monitor and a vital signs health monitor.

The Pulse Monitoring application (PBM) is an API that communicates with other autonomic components and the external environment. Pulse Monitoring is extending the HBM construct. HBM essentially only checks whether hosts are providing a 'heartbeat' or not. The lack of heartbeat will alert the designated controller that the system has died. Besides, checking whether the system is 'alive' or not, the PBM also indicates the health level (using NASA Beacon Monitors descriptors; NOMINAL, INTERESTING, IMPORTANT, or URGENT) of the system. The architecture of Globus Grid HBM is hierarchical, HBMLM (Local Monitor) reports to the HBMDC (Data Collector); the HBM is a client/server relationship, the HBMDC must be always available such that the HBMLM can report to it. While PBM is peer-to-peer, all hosts have equivalent capabilities and responsibilities. They are monitoring each other with minimal human interaction. A host has its autonomy to register & un-register with other hosts. Two hosts become neighbours after they register to each other. There is no limit on how many neighbour(s) that a host can register with. Hosts send Pulses to each other only when they are connected. Therefore, a host does not necessarily always have to be available, as a registered host may find another peer.

Pulse Monitoring self-healing tool contains four components; Main-Monitor, Internal-Monitor, External-Monitor, and Health-Monitor. Health-Monitor and Internal-Monitor monitor processes on a machine. Health-Monitor can re-start or terminate a failed process. External-Monitor communicates with the external environment, it sends/receives Pulses to/from other hosts (neighbours/peers); monitoring neighbours by sending a message to check if the neighbour is 'alive' or not when it detects that the neighbour hasn't sent its pulse and reboot a neighbour when necessary. Conversely, the host is being monitored by its neighbours in the same way. Main-Monitor is responsible for monitoring Internal-Monitor and External-Monitor. Main-Monitor would re-start them if they are 'dead'.

The aims of this proof of concept have been achieved. As a tool the Self-Healing prototype could be expanded in many ways. For instance, the Health-Monitor is now Windows platform specific, it could be extended to run on other operating systems, such as Unix or Linux etc. Each operating system has its own terminology on processes; the way an OS controls its processes can vary. The Health-Monitor could be extended to detect which operating system it is running on and to call the corresponding function to obtain process information.

Also the knowledge about a process in the Internal-Monitor is essentially *start* and *terminate*. The Internal-Monitor could be enhanced to automatically install/un-install a process (or application). This would need a knowledge base storing specific information and procedures of how to install/un-install each program.

Further autonomic options could evolve from the environment knowledge gained by the tool; for example, the ability to spot a process running intermittently or unstably. It may have a history of failing after running for a certain period of time on some executions. In this case, the process (the application) may need re-configuration or re-installation in order to run smoothly, in effect providing options for self-configuring and self-optimising and in so doing preventing the system degrading further (thus providing proactive self-protection and self-healing).

Acknowledgements

This work was undertaken through the Centre for Software Process Technologies, which is supported by the EU Programme for Peace and Reconciliation in Northern Ireland and the Border Region of Ireland (PEACE II).

References

[1] D. Garlan, J. Kramer, A. Wolf, (eds.) Proc. Workshop on Self-healing Systems (WOSS'02), ACM Press, Charleston, South Carolina, Nov. '02

[2] IBM, "An architectural blueprint for autonomic computing", April 2003

[3] R. Sterritt, D. Bustard, "Autonomic Computing-a Means of Achieving Dependability?", Proc. IEEE Int. Conf. on the Engineering of Computer Based System (ECBS'03), Huntsville, Alabama, USA, April 7-11 2003

[4] D. F. Bantz, C. Bisdikian, D. Challener, J. P. Karidis, S. Mastrianni, A. Mohindra, D. G. Shea, M. Vanover, "Autonomic personal computing", IBM Systems Journal, Vol 42, No 1, 2003

[5] R. Sterritt, "Pulse Monitoring: Extending the Health-check for the Autonomic GRID", IEEE Workshop on Autonomic Computing Principles and Architectures (AUCOPA' 2003) in Proc. IEEE Int. Conf. Industrial Informatics (INDIN 2003), Banff, Alberta, Canada, 22-23 August 2003.

[6] R. Sterritt, "Towards Autonomic Computing: Effective Event Management", Proceedings of the 27th Annual IEEE/NASA Software Engineering Workshop, Greenbelt, MD, Dec. 2002

[7] R. Sterritt, DW. Bustard, "Towards an Autonomic Computing Environment" 1st Int. Workshop on Autonomic Computing System in IEEE Workshop Proc. 14th Int. Conf. on Database and Expert Systems Applications (DEXA'2003), Sept 1-5 2003

[8] R. Sterritt, D. Gunning, A. Meban, P Henning, "Exploring Autonomic Options in an Unified Fault Management Architecture through Reflex Reactions via Pulse Monitoring", IEEE Workshop on the Engineering of Autonomic Systems (EASe 2004) in Proc. 11th Ann. IEEE Int. Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2004), Brno, Czech Republic, 24-27 May 2004

[9] A. G. Ganek, T.A. Corbi, "The dawning of the autonomic computing era", IBM Systems Journal, Vol 42, No 1, 2003

[10] J. Appavoo, K. Hui, C.A.N. Soules, R.W. Wisniewski, D.M. Da Silva, O. Krieger, M.A. Auslander, D.J. Edelsohn, B. Gamsa, G.R. Ganger, P. McKenney, M. Ostrowski, B. Rosenburg, M. Stumm, and J. Xenidis, Enabling autonomic behavior in systems software with hot swapping, IBM Systems Journal, Vol 42, No 1, 2003, pp. 60-76

[11] What is P2P, http://compnetworking.about.com/library/weekly/aa093000a.htm

[12] Peer-to-Peer Computing is Good Business, http://www.intel.com/eBusiness/products/peertopeer/ar010102.htm

[13] Peer-to-Peer Architecture, http://80211-planet.webopedia.com/TERM/p/peer_to_peer_architecture.html

[14] The Globus Heartbeat Monitor Specification v1.0, http://www-fp.globus.org/hbm/heartbeat_spec.html

[15] E. Jay Wyatt, Henry Hotz, Robert Sherwood, John Szijjarto, Miles Sue, "Beacon Monitor Operations on the Deep Space ONE Mission", Jet Propulsion Laboratory, California Institute of Technology

[16] D. DeCoste, S. G. Finley, H. B. Hotz, G. E. Lanyi, A. P. Schlutsmeyer, R. L. Sherwood, M. K. Sue, J. Szijjarto, E. J. Wyatt, "Beacon Monitor Operations Experiment DS1 Technology Validation Report", Jet Propulsion Laboratory, California Institute of Technology

[17] E. J. Wyatt, M. Foster, A. Schlutsmeyer, R. Sherwood, M. K. Sue, "An Overview of the Beacon Monitor Operations Technology", Jet Propulsion Laboratory, California Institute of Technology

[18] Naik VK, Sivasubramanian S, Bantz DF, "Harmony: A Desktop Grid for Delivering Enterprise Computations," Proceedings of the 4th Int. Workshop on Grid Computing (Grid 2003), Phoeniz, Arizona, November 2003

[19] Microsoft Platform SDK Documentation, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/perfmon/base/about_performance_monitoring.asp

[20] "Java Programming Language", Sun Microsystems, Inc., Enterprise Services Jan 1999, Revision B.1

[21] Mike Pietraszak, "Using J/Direct to Call the Win32 API from java", http://www.microsoft.com/mind/0198/default.asp

[22] Joseph L. Weber, M. Wutka, "Using Java 2" Publisher: Que Publishing

[23] J. Reynolds, J. Postel, "RFC 1700 – Assigned Number", Network Working Group