Técnicas para mejorar la confiabilidad del software

Por Ing. Manuel Rodríguez Echevarría e Ing. Frank Niz Fernández
Especialistas Unidad de Negocios Red y Unidad de Negocios Clientes, ETECSA
manuel.rodriguez@etecsa.cu, frank_unc@etecsa.cu

Introducción

El desarrollo alcanzado por la industria informática ha conllevado una notable reducción en los porcentajes de averías del hardware, debido a las continuas mejoras tecnológicas. En este entorno, el elemento clave en disponibilidad de los sistemas ha pasado a ser la confiabilidad del software. Hoy día, la caída de un sistema se debe más al resultado de un fallo de software que de un fallo de hardware. A pesar de los esfuerzos para eliminar los defectos (*bugs*) antes de desplegar estos sistemas, es prudente asumir que los errores siempre existirán y casi siempre conducirán a un fallo.

La importancia creciente del software en la confiabilidad de los sistemas informáticos

Los sistemas de software, por el hecho de ser productos de la creación humana, son vulnerables a los errores de programación. Estos errores ocurren en varias formas, que incluyen las inconsistencias de diseño, los errores de sintaxis y los semánticos. En los sistemas de uso común, estos defectos causan inconvenientes y molestias; pero no son particularmente serios. Sin embargo, resultan muy graves para los sistemas donde la seguridad es crítica, en los cuales un fallo puede causar la muerte, daños o enfermedades a las personas, grandes pérdidas económicas, fracaso de misiones, afectaciones al medio ambiente, conflictos armados, etc. [1, 2].

Los costos asociados a los fallos de software, además de la pérdida económica directa, también incluyen los gastos para corregir los defectos que causaron el fallo después de que el sistema está implementado, y que se estima en 10 mil USD/defecto, principalmente por el tiempo requerido por un programador para encontrar y arreglar el problema en el código [2, 3].

Se ha observado que la densidad promedio de defectos en un módulo de programa —errores/línea— se mantiene más o menos constante para diferentes lenguajes de programación. Desde un punto de vista fisiológico, esto significa que el programador tiene una probabilidad constante de introducir defectos, por lo tanto, si se duplica el tamaño de un módulo software, este contendrá el doble de defectos [2, 4]. Para contrarrestar lo anterior, generalmente, se aplica la técnica **divide y vencerás**, con la fragmentación del software en varios módulos de tamaño manejable. Sin embargo, el proceso de construir software a partir de **cajas negras** tiene el potencial de introducir más defectos. El problema reside en las interfaces entre módulos, presentes dondequiera que un módulo interactúe con otro [5]. Mediante investigaciones empíricas en software real, se ha encontrado que los defectos en las interfaces son proporcionales al número de interacciones, es decir, N^x para N módulos, donde el exponente varía entre x = 1,2 para software alfa, en vías de desarrollo, y x = 1,1 para versiones maduras [6].

Según un estudio realizado por la NASA en el año 2002, el tamaño del software utilizado en sus misiones espaciales se ha incrementado exponencialmente durante los últimos años [7]. Entonces, es de esperar que en el futuro próximo se produzca un crecimiento notable de fallos a causa de los defectos inherentes al software, a menos que se apliquen otras medidas para neutralizar esta tendencia.

Clasificación general de los defectos inherentes al software [8, 9, 10]

Defectos determinísticos (*bohrbugs*): idealmente debieron haber sido eliminados durante la fase de depuración (*debugging*); pero a pesar de que el software haya sido exhaustivamente probado, pueden quedar ocultos algunos errores de diseño, incluso en programas maduros, por ejemplo, los sistemas operativos de uso comercial, sobre todo cuando están compuestos de partes viejas y partes relativamente nuevas asociadas con actualizaciones, nuevas funciona-

lidades, etc. La única forma de neutralizarlos es mediante la diversidad de diseños, en la cual se utilizan varias aplicaciones que proveen la misma funcionalidad pero con diferentes implementaciones.

Defectos aleatorios o de incertidumbre (*heisenbugs*): son defectos en el software que sólo se manifiestan durante coincidencias específicas de eventos. Por ejemplo, algunas operaciones pueden dejar el software en un estado que provoque un error en una operación ejecutada a continuación; otro ejemplo es la pérdida de sincronización entre los hilos de un proceso, donde los errores ocurren durante algunas ejecuciones, pero no ocurren cuando se repite la ejecución. Se dice que estos errores son causados por fallas transitorias. El problema puede ser resuelto con el reintento de la operación o el reinicio del proceso, si este ha caído.

Defectos asociados al envejecimiento (*aging-related bugs*): los recursos como el espacio para *swap*, la memoria libre disponible, etc. se degradan de manera progresiva por defectos en el software —pérdida de memoria y liberación incompleta de recursos después de su uso—. La estimación de la tasa de agotamiento de los recursos y, por lo tanto, el momento esperado de la falla, han sido los focos de las investigaciones sobre rejuvenecimiento del software. Según este principio, la caída de un sistema puede evitarse mediante el reinicio periódico de un proceso o nodo, según una predicción basada en las mediciones de la tasa de agotamiento de determinados recursos.

Enfoques y métodos orientados a mejorar la confiabilidad del software

En general se usan dos grandes enfoques que se complementan: el primero, orientado a minimizar los errores de programación y, el segundo que, con la aceptación de los defectos inevitables, se concentra en el diseño de sistemas inmunes a ellos. En la figura 1 se muestra una panorámica general de las características de ambos [2].

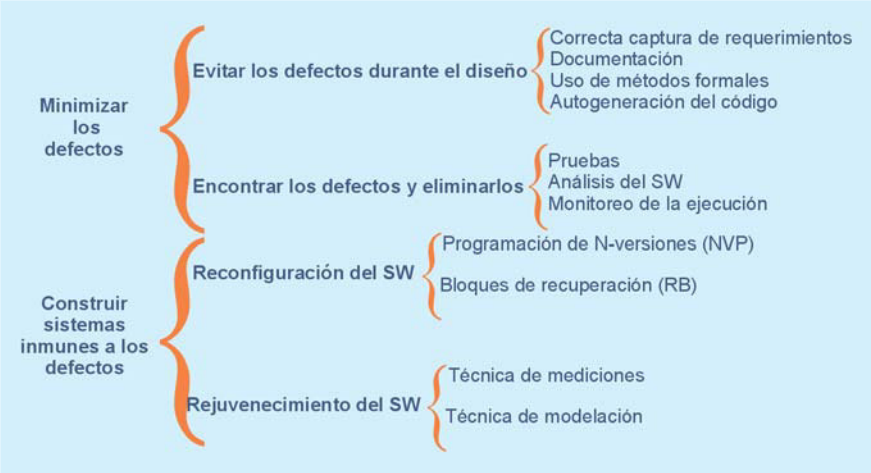

Figura 1 Enfoques para mejorar la confiabilidad del software

Enfoque 1 —minimizar los defectos—

El desarrollo de metodologías formales y herramientas para el diseño de software ha contribuido notablemente a la disminución de la tasa de errores en el producto final, aunque nunca se llega a su eliminación total.

Enfoque 2 —construir sistemas inmunes a los defectos—

Para que un sistema resulte inmune a los defectos, debe ser capaz de recuperarse por sí mismo ante la ocurrencia de un fallo (reconfiguración) o mantenerse en un estado de funcionamiento que evite que los defectos latentes lleguen a manifestarse (rejuvenecimiento).

Reconfiguración del software

El campo de los sistemas tolerantes a fallas ha evolucionado debido a la necesidad de construir sistemas suficientemente confiables a partir de dispositivos físicos poco confiables. El éxito de las técnicas tolerantes a fallos en el campo del hardware sugirió que un enfoque similar sería útil para enfrentar los errores del software [2]. En este sentido se han destacado dos técnicas de reconfiguración del software que constituyen diferentes implementaciones de un mismo concepto: utilización de varias versiones de código para realizar la misma tarea. Estas técnicas son:

- Programación de N versiones —N-*Version Programming* (NVP)—
- Bloques de recuperación —*Recovery Blocks* (RB)—

La programación de N versiones fue propuesta por primera vez en 1977 y, como su nombre sugiere, se basa en disponer de varias (N > 2)

versiones de programas equivalentes basadas en las mismas especificaciones [1, 11]. Esta técnica utiliza la redundancia con un enfoque de replicación y votación. Las versiones deben ser desarrolladas por equipos de programadores independientes y, de ser posible, esos equipos deben usar diferentes algoritmos y lenguajes de programación para

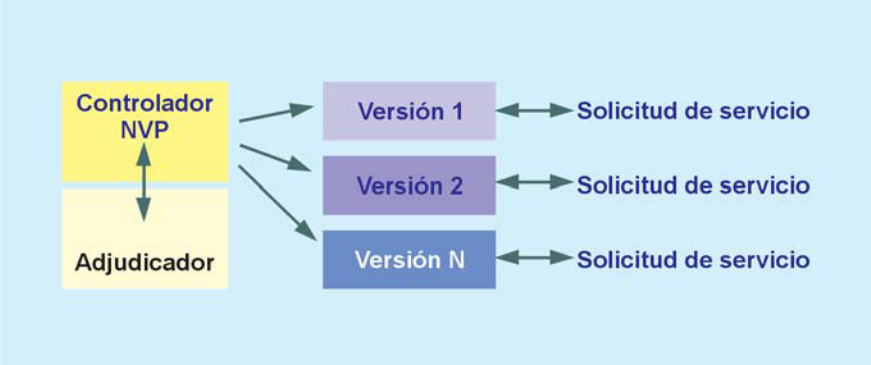

Figura 2 Programación de N versiones

minimizar la probabilidad de errores comunes. Las N versiones se ejecutan paralelamente por varios procesadores y, los resultados se comparan mediante una unidad de adjudicación que implementa un algoritmo de decisión (Figura 2).

Por otra parte, la técnica de los bloques de recuperación utiliza un enfoque dinámico de la redundancia. En cada momento se ejecuta un solo programa y, por lo tanto, no hay fuertes requerimientos de hardware. El enfoque de los RB (Figura 3) es similar al de la NVP en el sentido de que se escriben varias rutinas para un mismo problema. Sin embargo, la ejecución se realiza de manera diferente: antes de entrar a un RB se establece un punto de recuperación (RP1); entonces, corre el primer módulo (M1) y produce un resultado. Se realiza un test de aceptación, y la ejecución normal continúa si el test es satisfactorio; de lo contrario, se retorna el sistema al punto de recuperación y se invoca el segundo módulo (M2). Así, sucesivamente, continúa el proceso hasta que el test de aceptación resulte satisfactorio o se agoten las alternativas, en cuyo caso se genera una excepción.

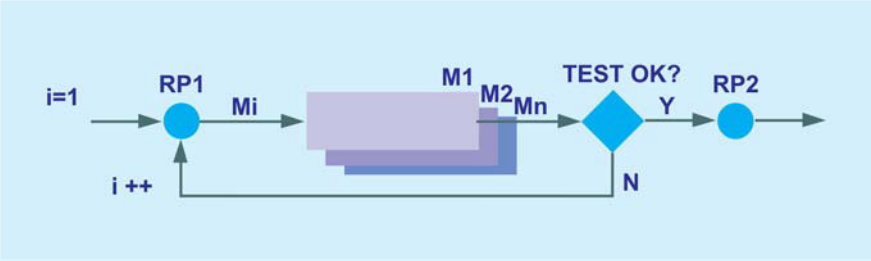

Figura 3 Bloque de recuperación

Rejuvenecimiento del software

Los defectos del software, cuando se activan, no siempre causan fallos inmediatos, sino llevan el sistema a un estado en el cual comienza a decaer. Este decaimiento tiene síntomas como la pérdida de memoria, ruptura de punteros, el bloqueo de ficheros que nunca se liberan, la acumulación de errores numéricos, etc. que acusan una degradación gradual en la disponibilidad del servicio y en la calidad de los datos y, eventualmente, pueden conllevar a la caída total del sistema. El rejuvenecimiento del software —*Software Rejuvenation* (SR)— es un enfoque proactivo que implica detener periódicamente la ejecución de un software, limpiar los estados internos y luego reiniciar el software. No elimina los defectos asociados al envejecimiento, sino evita que estos lleguen a manifestarse como fallos impredecibles del sistema.

En la figura 4*A* se muestra el diagrama de estados de un proceso sin rejuvenecimiento. A partir de un estado inicial S_0, el sistema se degradará progresivamente y alcanzará el estado de falla probable S_P. Si un defecto de software llega a manifestarse, se alcanzará el estado de fallo S_F y habrá que reiniciar el sistema para llevarlo al estado original. En la figura 4*B*, donde se utiliza rejuvenecimiento, puede apreciarse que desde el estado S_P es posible llevar oportunamente el sistema a su

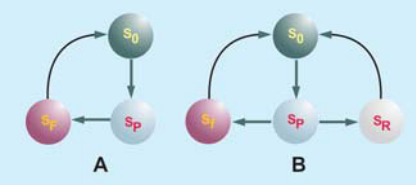

Figura 4 Estados del sistema —A: sin rejuvenecimiento, B: con rejuvenecimiento—

estado inicial a través del estado de rejuvenecimiento S_R, antes de que se produzca un fallo. En este caso, a diferencia del anterior, el momento de la indisponibilidad será previamente programado para que su costo resulte mínimo.

El rejuvenecimiento del software es tan intuitivo como reiniciar una PC de vez en cuando, pero ha sido aplicado en sistemas de software de gran escala [1, 12]. Es, además, similar al mantenimiento preventivo en los sistemas hardware:

implica que algunos servicios queden indisponibles temporalmente, la idea es evitar que, en el futuro, ocurran fallos inesperados de mayor duración. El uso de esta técnica puede incrementar la confiabilidad de un sistema en dos órdenes de magnitud [13].

El factor clave para lograr que la indisponibilidad programada sea preferible a la indisponibilidad inesperada está en determinar con qué frecuencia debe ser rejuvenecido un sistema. Si los fallos inesperados son catastróficos, entonces se justifica un calendario de rejuvenecimiento más agresivo en términos de costo y disponibilidad. Si los fallos inesperados tienen un costo similar a la indisponibilidad programada, entonces es más apropiado un enfoque reactivo. Generalmente, las dos técnicas utilizadas para determinar el ciclo óptimo de rejuvenecimiento son:

- La técnica de mediciones, que estima el momento del rejuvenecimiento a partir del estado de los recursos del sistema [14, 15].
- La técnica de modelación, que usa modelos matemáticos —cadenas de Markov, redes estocásticas de Petri— y simulación, para estimar el momento del rejuvenecimiento, basada en el rendimiento previsto [10, 16].

Estas técnicas son especialmente útiles durante la fase de operación del sistema, con vista a neutralizar los defectos remanentes del proceso de especificación del software [17]. A partir de la base teórica se han desarrollado varias herramientas que facilitan la modelación y simulación de estos procesos [18, 19, 20].

Campos de aplicación de las técnicas para elevar la confiabilidad del software

El enfoque orientado a minimizar los defectos y el diseño de software reconfigurable, sólo son aplicables en un entorno de producción; sin embargo, no son accesibles para quienes utilizan software previamente diseñado. Un ejemplo clásico son los sistemas operativos Windows, de utilización mundial, y de los cuales solamente el fabricante tiene acceso al código fuente. También es cierto que las técnicas antes mencionadas se justifican cuando el costo de un fallo inesperado es muy superior al costo de un fallo programado. Por lo tanto, en la mayoría de las estaciones de trabajo donde se ejecutan procesos de oficina en tiempo diferido, puede ser recomendable un enfoque reactivo —reiniciar el sistema si falla—.

En general, las empresas cuentan con planes para el mantenimiento de su hardware instalado; pero no es común que se realice un mantenimiento proactivo de las aplicaciones y sistemas operativos. Aunque en los diagnósticos de seguridad informática usualmente se identifica el concepto **fallo de programa** como un riesgo, las medidas que se proponen son, generalmente, reactivas, orientadas a establecer bancos de programas y aplicaciones para ejecutar reinstalaciones en caso de fallos. Específicamente, para las empresas de telecomunicaciones resultan de particular interés el posible uso de los productos de reconfiguración disponibles en el mercado para las estaciones de trabajo directamente vinculadas a los procesos de gestión de la red y los servicios, y el rejuvenecimiento del software en algunos servidores donde corren procesos críticos para el mantenimiento de los servicios de telecomunicaciones.

Conclusiones

Aunque existen dos enfoques fundamentales encaminados a elevar la confiabilidad del software, en un entorno de diseño, estos métodos deben combinarse para enfrentar la creciente complejidad de los sistemas; pero en el caso práctico de los **usuarios** de software, resultan de interés particular las técnicas de rejuvenecimiento.

Bibliografía

[1] Yurcik, William; Doss, David. "Achieving Fault-Tolerant Software with Rejuvenation and Reconfiguration". *IEEE software*, Vol.18 no.4 (July- August 200): 48-52. Disponible en: http://www.ee.pdx.edu/~greenwd/software_rejuvenation.pdf (Consultado: diciembre de 2005).

[2] Dimond, Robert; Madhvani, Neil; Mathai, Jonathan. "Software for NASA in 2050: an Impossible Mission?" Imperial College of Science, Technology & Medicine. Department of Electrical and Electronic Engineering & Department of Computing, 2002
Disponible en: http://www.matcore.com/surprise/report/Report_FINAL.pdf (Consultado: diciembre de 2005).

[3] Bowen, Jonathan; Stavridou, Victoria. "Safety-Critical Systems, Formal Methods and Standards". *IEEE Software Engineering Journal*, Vol.8 no.4 (July 1993): 189-209.

[4] Hatton, Les. "Software Failures Follies and Fallacies". *IEEE Review*, Vol.43 no.2 (March 1997): 49-51.

[5] Hatton, Les. "Reexamining the Fault Density-Component Size Connection". *IEEE Software*, (March-April 1997): 89-96.
[6] Boehm, Barry. "Quantitative Evaluation of Software Quality", Proceedings, Second International Conference on Software Engineering, 1976, pp 592-605.
[7] Lowry, Michael R. "Software Construction and Analysis Tools for Future Space Missions". 8th International Conference, TACAS 2002. Vol. 2280Pub. Springer/Verlag. Berlin Heidelberg 2002, pp 1-19.
[8] Gray, Jim. "Why do Computers Stop and What Can Be Done About it?", Proc. of 5th Symp. on Reliability in Distributed Software and Database Systems. January, 1986, pp 3-12.
[9] Software Bugs. Disponible en: http://www.srel.ee.duke.edu/software_bugs.htm (Consultado: diciembre de 2005).
[10] Dong Chen, S. Dharmaraja, Dongyan Chen, Lei Li, Kishor S. Trivedi, Raphael R. Some, Allen P. Nikora. "Reliability and Availability Analysis for the JPL Remote Exploration and Experimentation System". Proceedings of the International Conference on Dependable Systems and Networks (DSN'02). IEEE, 2002. Disponible en: http://www.ee.duke.edu/~kst/JPL/DSN2002.pdf. (Consultado: diciembre de 2005).
[11] Avizienis A. "The Methodology of N-Version Programming". En: *Software Fault Tolerance*. Editado por Lyu M. R. New York: John Wiley & Sons,1995, pág. 23-46.
[12] Kishor S. Trivedi, Kalyan Vaidyanathan "Software Rejuvenation: Modeling and Analysis" TOOLS 2003, Tutorial September 2, 2003 Urbana, IL, USA. Disponible en: http://www.crhc.uiuc.edu/Multi/Vaidyanathan-2003IllinoisMulticonference.pdf (Consultado: diciembre de 2005).
[13] Lawrence Bernstein, Chandra M. R. Kintala, "Software Rejuvenation" Stevens Institute of Technology.
[14] Kenny C. Gross, Vatsal Bhardwaj, Randy Bickford. "Proactive Detection of Software Aging Mechanisms in Performance Critical Computers" Sun Microsystems, Duke University, Expert Microsystems. Disponible en: http://www.ee.duke.edu/~vb/grossK_softaging.pdf (Consultado: diciembre de 2005).
[15] Vittorio Castelli, Rick Harper, et al. "Software rejuvenation". Disponible en: http://www.research.ibm.com/people/r/reharper/2003%20PFA%20Conference/22-softwarer%20academy%20pres.ppt (Consultado: diciembre de 2005).
[16] Héctor Enrique Gaona-Flores, Víctor Sánchez González, Gerardo Juárez Flores, Rene Díaz Martínez "Análisis de modelos matemáticos para el rejuvenecimiento del software usando la metodología regenerativa de Markov y las redes estocásticas de Petri". Universidad Autónoma del Estado de México, Unidad Académica Profesional Valle de Chalco, Instituto Superior Politécnico José Antonio Echeverría, La Habana, Cuba.
[17] SENG 609.31 - Introduction to Dependability of Computing Systems: Commentary on "Achieving Fault-Tolerant Software with Rejuvenation and Reconfiguration by William Yurcik and David Doss". Disponible en: http://64.233.179.104/search?q=cache:4nuyb4l3JTMJ:sern.ucalgary.ca/~lawa/SENG60931/read/read2.doc+yurcik+doss+rejuvenation&hl=es (Consultado: diciembre de 2005).
[18] SHARPE (Symbolic Hierarchical Automated Reliability and Performance Evaluator). Disponible en: http://www.ee.duke.edu/%7Echirel/IRISA/sharpeGui.html (Consultado: diciembre de 2005).
[19] SPNP (Stochastic Petri Net Package). Disponible en: http://www.ee.duke.edu/%7Echirel/IRISA/spnp.html (Consultado: diciembre de 2005).
[20] SREPT (Software reliability estimation and prediction tool). Disponible en: http://www.ee.duke.edu/%7Ekst/srept_description.html (Consultado: diciembre de 2005).

Nota: en este artículo se ha decidido hacer una excepción en relación con las normas para citas, notas o referencias bibliográficas y la bibliografía específicas de la revista. Por su particularidad, se ha respetado la forma en que las han utilizado los autores.