Módulo de recuperación de fallos de los procesos de un sistema FreeBSD utilizando variables en disco

Javier Vargas[1], David Guevara[1], Franklin Mayorga[1], Ernesto Jiménez[2]

[1] Facultad de Ingeniería en Sistemas, Electrónica e Industrial, Universidad Técnica de Ambato, Av. Los Chasquis y Río Payamino, Ambato, Ecuador.

[2] Universidad Politécnica de Madrid, España.

Autores para correspondencia: js.vargas@uta.edu.ec, dguevara@uta.edu.ec, fmayorga@uta.edu.ec, ernes@eui.upm.es

Fecha de recepción: 17 de mayo 2017 - Fecha de aceptación: 2 de agosto 2017

ABSTRACT

This paper describes the creation of a crash-recovery module of the processes in a reliable communication program with fall and recovery of equipment, on a FreeBSD system. The creation of this module is based on the parameters or values that a running process contains, establishing a variable or file that is stored on disk. The variable is synchronized with the values of each process so that they can be retrieved by the module, avoiding loss of information during the transmission prior to the data integration. The module presents two recovery mechanisms, the coherence checker and the concurrency control, each of which process in a synchronized way with the process and the recovery of the variable in the event of a failure.

Keywords: crash-recovery module, recovery mechanisms, coherence checker, concurrency control.

RESUMEN

El presente trabajo describe la creación de un módulo de recuperación de fallos de los procesos de un programa de comunicación fiable con caídas y recuperación de equipos, en un sistema FreeBSD. La creación de este módulo se basa en los parámetros o valores que un proceso en ejecución contenga, estableciendo una variable o fichero que se almacena en disco. La variable se sincroniza con los valores de cada proceso para que puedan ser recuperados por el módulo, con la finalidad de que no se pierda la información o datos que se estén trasmitiendo al momento de reintegrarse. El módulo presenta dos mecanismos de recuperación, el comprobador de coherencia y el control de concurrencia, cada uno de ellos trabaja de forma sincronizada con el proceso y la variable de recuperación en caso de presentarse algún tipo de fallo.

Palabras clave: módulo de recuperación, mecanismos de recuperación, comprobador de coherencia, control de concurrencia.

1. INTRODUCCIÓN

La amplia información que se trasmite en un sistema distribuido de equipos es gestionada por los procesos que realizan este tipo de tareas de envío y recepción de datos (Kamoshida & Taura, 2008). Básicamente un bloque de datos se transmite por la capa de transporte del cuarto nivel del modelo OSI asegurando el envío y recepción libre de errores (Chowdhury & Mustafa, 2010), sin embargo, no contiene un módulo de recuperación ante posibles fallos. Un error o fallo comúnmente se presenta en el equipo que se encuentra comunicando con otro, estos fallos pueden ser por incidencias de parada forzada o terminación del proceso (Pina & Hicks, 2016). Si el proceso está en relación con el sistema, los estados básicos que tiene son: ejecución, listo y bloqueo (Jung, Hyuh & Ma, 2001), pero no de recuperación.

Por otro lado, si el programa falla y nuevamente quiere reintegrase, el proceso debe conocer los parámetros necesarios para poder recuperar su funcionamiento anterior; esto en un sistema de tolerancia a fallos a nivel de proceso como se ilustra en la Figura 1.

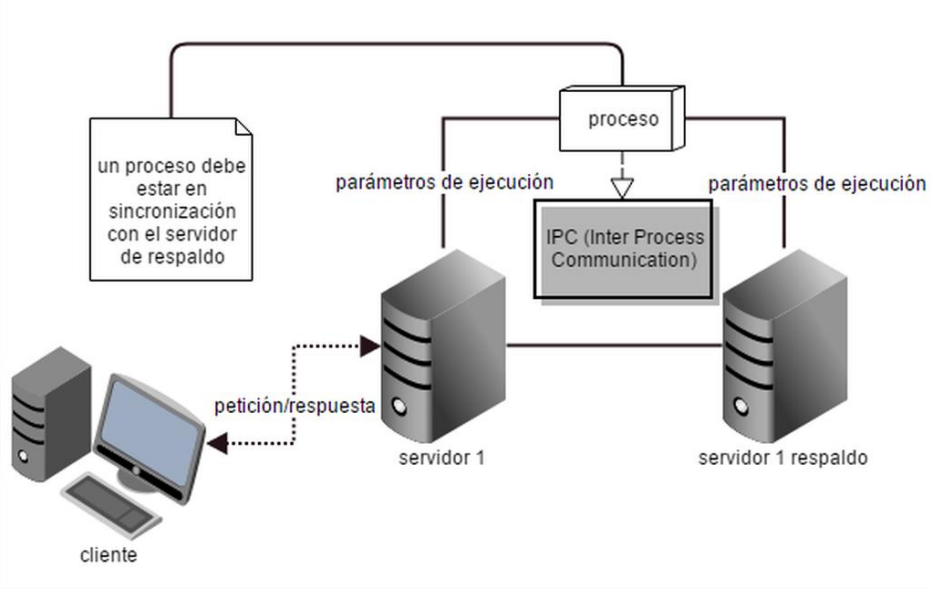

Figura 1. Sistema tolerante a fallos con respaldo.

Actualmente existen herramientas o mecanismos que permiten el dinamismo en recuperación y detección de fallos. Para la recuperación se tiene un comprobador de coherencia (Abts, Roberts & Lilja, 2000), que compara la distribución de los directorios con cada bloque de datos del proceso, y para la detección de fallos se tiene el control de concurrencia (Sutter, 2005), que busca cambios o errores producidos en el bloque de datos que se trasmite. Una de estas herramientas es Spread Toolkit (2017) la cual utiliza una variedad de aplicaciones de código abierto bajo una licencia del estilo BSD (Berkeley Software Distribution), con el objetivo principal de comunicar y sincronizar los diferentes procesos del sistema en redes que son propensas a fallar (Mitterbauer, Zeilinger, Kohlhauser, Turek & Kreilmeier, 2011). La fiabilidad de que el programa o sistema siga funcionando dependerá del módulo de recuperación ante fallos gestionables por una variable o fichero en disco.

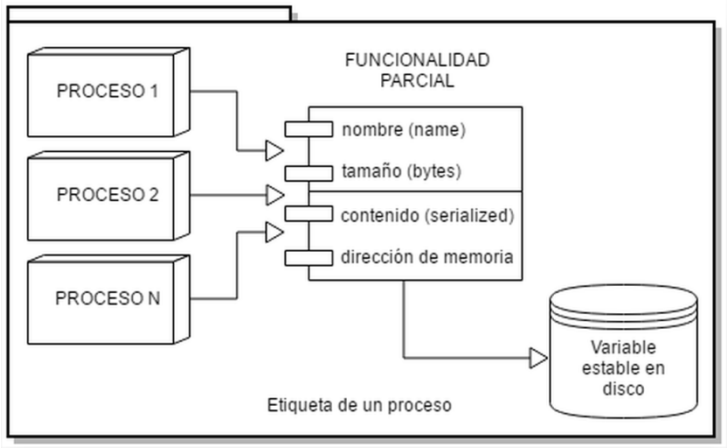

Figura 2. Etiquetas de procesos almacenados en disco.

La división de este módulo se basa en dos etapas, detección y recuperación. En un ámbito de gestión de sistemas distribuidos en FreeBSD (McKusick, Neville-Neil & Watson, 2014), la recuperación de los posibles errores que se puedan presentar se consigue con la funcionalidad parcial de relación que asocia al proceso en ejecución con sus valores o parámetros de un proceso. Un proceso con funcionalidad

parcial guarda los valores de ejecución en una variable en disco, conteniendo los valores del nombre, tamaño, contenido y dirección de memoria, permitiendo así la sincronización del proceso con el sistema en ejecución (Claus, Zeppenfeld, Müller & Stechele, 2007). La etiqueta de un proceso se utiliza para diferenciar entre las variables almacenadas de recuperación con varios procesos en ejecución como se ilustra en la Figura 2. Para lo cual, se presenta un módulo de recuperación de fallos de los procesos del sistema, diferenciando de Spread Toolkit con la utilización de variables en disco, además de integrar los dos mecanismos que son el comprobador de coherencia y el control de concurrencia.

2. MATERIALES Y MÉTODOS

Los valores de cada proceso se registran en diferentes clases en el comprobador de coherencia y cada una de ellas por encadenamiento son gestionadas en el control de concurrencia, empaquetando así el módulo de recuperación de fallos; la principal característica que se presenta es el manejo de hilos entre clases con boost.thread que permite el uso de múltiples subprocesos de ejecución con datos compartidos (Trott & Olson, 2010).

Las clases son creadas en orden de fundamento:

- registerVariable: La variable no se almacenará en un almacenamiento estable hasta que se active explícitamente la clase synchronize por primera vez.
- synchronize: Actualiza el valor de una variable en un almacenamiento estable.
- recoverVariable: Recupera el valor de una variable de un almacenamiento estable, además localiza la variable de recuperación existente o se crea una nueva variable si no la encuentra, locateRecoveryFile y loadRecoveryData.

2.1. Obtención de los valores del proceso que falle (processTag)

La recuperación de un proceso en ejecución de un equipo que fallase debe acceder a los datos o al recurso necesario para poder reintegrarse y volver a estar en funcionamiento. Cabe recalcar que estas caídas o fallos son de incidencias poco conocidas (Angela, Granizo, Alex & Tacuri, 2016), error en red, detención forzosa, etc. Un proceso que falle y quiera reintegrarse debe esperar un cierto tiempo que se delimita por el tamaño de dirección de memoria que estuviese consumiendo. Para lo cual se almacenan estos parámetros:

- name: Variable identifier (nombre que identifica la variable).
- size: Variable data size (tamaño variable en bytes).
- needsSerialization: Variable bool (la variable tiene que ser serializada / deserializada).
- appVariable: Application variable memory address (puntero a la variable).

Estos valores son registrados por la función de la biblioteca de C memcpy (Merz, Falke & Sinz, 2012), que copia los valores del área de memoria para después ser utilizados en la clase registerVariable como se ilustra en la Figura 3.

2.2. Registro de los valores de la variable en disco (registerVariable)

La clase que registra los valores debe contener un cierto grado de sincronización con el proceso en ejecución. Primero se actualiza el valor de una variable en un almacenamiento estable, se debe registrar en el módulo de que se trata de una operación síncrona con el método thread-safe (seguridad en hilos) (Kleiman, Shah & Smaalders, 1996). Posteriormente el puntero a la variable entrará a la clase synchronize, el cual debe comprobar con un método de excepción que no exista un error de estado del proceso (Dickens & Thakur, 1999), como se ilustra en la Figura 3.

```
#ifdef DEBUG_CRM
    std::cerr << "[CRM] registerVariable("
                                        << name << ",size=" << size << ")" << std::endl;
#endif // DEBUG_CRM
    if (name.size() > MAX_VARIABLE_NAME)
        throw new CrashRecoveryException("registerVariable: variable name too long");
    for (auto variable : variables)
        if (variable->name == name)
            throw new CrashRecoveryException("registerVariable: attempt to" +
                                             +"register a duplicate variable name");
    variable = new StableVariable;
    variable->name = name;
    variable->size = size;
    variable->appVariable = appVariable;
    variable->data = new char[size];
    memcpy(variable->data, appVariable, size); // Not strictly necessary
    variables.push_back(variable);
```

Figura 3. Extracto de script de CrashRecoveryException/registerVariable.

2.3. Sincronización de los valores del proceso con la variable en disco (synchronize)

Trabajar de forma síncrona dentro de esta clase permite la ejecución en tiempo real para guardar el tipo de proceso, dependiendo del puntero a la variable, appVariable. Si la variable no necesita serialización, se trata de un puntero directo a la variable de proceso real. Por otro lado, si la variable necesita deserialización se crea una nueva instancia en memoria de la variable y este parámetro se considerará como un puntero indirecto que se actualizará al volver (Plow, 1983), recoverVariable como se ilustra en la Figura 4.

```
#ifdef DEBUG_CRM
    std::cerr << "[CRM] synchronize(";
#endif // DEBUG_CRM
    for (auto variable : variables) {
        if (variable->appVariable == appVariable) {
#ifdef DEBUG_CRM
            std::cerr << variable->name;
            if (variable->size == sizeof(int))
                std::cerr << "," << *((int*) variable->appVariable);
            std::cerr << ")" << std::endl;
#ifdef DEBUG_CRM
    std::cerr << "[CRM] recoverVariable(" << name << ")";
#endif // DEBUG_CRM
    for (auto variable : variables) {
        if (variable->name == name) {
            variable->appVariable = appVariable;
            memcpy(appVariable, variable->data, variable->size)
#ifdef DEBUG_CRM
            if (variable->size == sizeof(int))
                std::cerr << " = " << *((int*) variable->data);
            std::cerr << std::endl;
#endif // DEBUG_CRM
            return true;
```

Figura 4. Extracto de script de appVariable/synchronize.

2.4. Recuperación de los valores del proceso con la variable en disco (recoverVariable)

La recuperación de los valores se define como funcionalidad parcial, asociando todas las clases del proceso desde la obtención de los valores, hasta la sincronización en memoria que se almacena posteriormente en disco. Los métodos locateRecoveryFile y loadRecoveryData confirman que se carguen los valores de las variables almacenadas en la memoria desde su variable de recuperación, para que la clase sincronice ejecute las comprobaciones para asegurar que la variable sea coherente y que ningún otro proceso local la esté utilizando. La búsqueda de la variable que coincida con el filtro del nombre debe reintegrase tan pronto se pueda abrir la variable válida sin propietario, como se ilustra en la Figura 5. La etiqueta processTag, se utiliza para comprobar si este variable de recuperación pertenece al tipo de proceso en ejecución.

```
// If the descriptor is valid, load the variable contents.
if (numBytesRead == sizeof(vdesc) && vdesc.valid) {
    StableVariable* variable = 0;
    for (auto v : variables) {
        if (v->name == vdesc.name) {
            variable = v;
            break;
        }
    }
    if (variable == 0) {
        variable = new StableVariable;
        if (vdesc.name[sizeof(vdesc.name) - 1] == '\0')
            variable->name = string(vdesc.name);
        else
            variable->name = string(vdesc.name, vdesc.name
            + sizeof(vdesc.name) - 1);
        variable->size = vdesc.size;
        variable->appVariable = 0;
        variable->data = new char[variable->size];
        variables.push_back(variable);
    }
```

Figura 5. Extracto de script de variable/loadRecoveryData.

3. RESULTADOS Y DISCUSIÓN

La comprobación del módulo de recuperación de fallos de los procesos del sistema utilizando variables en disco, se lo realiza en un programa (FDDemo) creado para un sistema de comunicación fiable. El sistema permite la ejecución de procesos para la comunicación entre varias instancias de una red. El módulo crea por cada instancia una variable en disco la cual se sujetará a los valores de las etiquetas del proceso, de la Sección 2.1. En la Figura 6 se muestra el funcionamiento del programa FDDemo, el cual presenta 3 instancias de red en un sistema distribuido de equipos, cada instancia almacena una variable en disco (extensión .rec). Esta variable en disco permite al módulo de recuperación saber cuál es el proceso en ejecución que se detuvo y que debe reintegrarse.

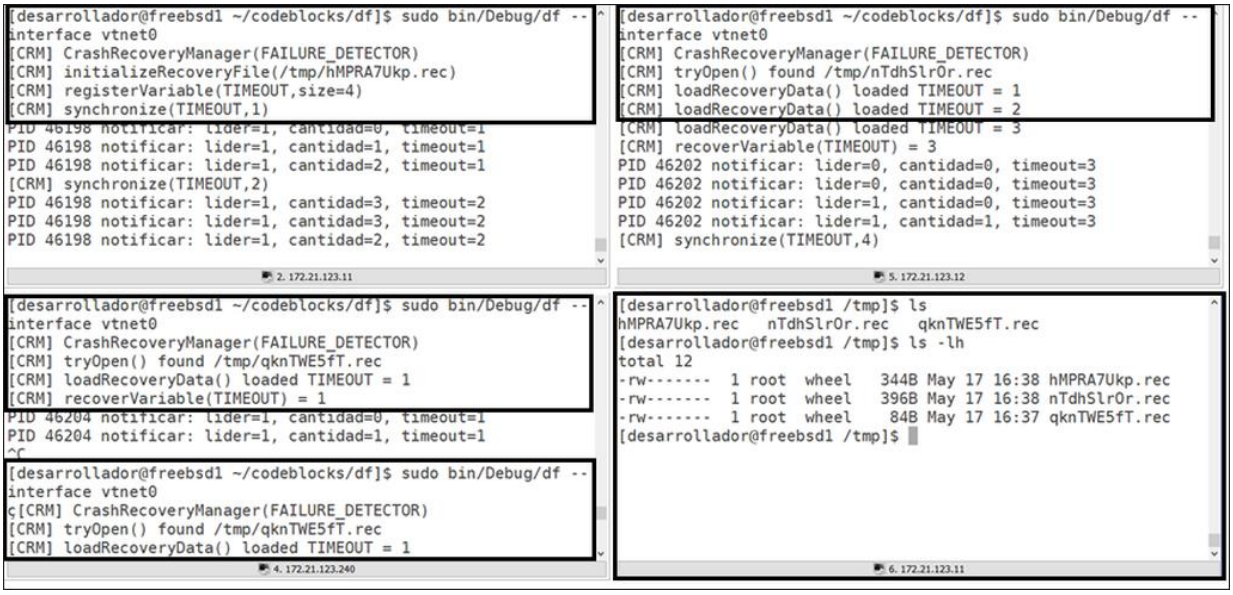

Figura 6. Funcionamiento del programa FDDemo.

La variable en disco (.rec) tiene los parámetros de ejecución por cada proceso de las instancias creadas, primero se genera un volcado en binario de la variable (00000000), a continuación se tiene el contenido en hexadecimal (43 52 41 53 48 2d 52 45 43 4f 56 45 52 59 30 31), todo esto para diferenciar en el programa los valores por cada variable (CRASH-RECOVERY01) de recuperación, de la Sección

2.2. Por otro lado, también se almacena el identificador del tipo de proceso que pertenece esa variable en disco (FAILURE_DETECTOR) con la misma secuencia de datos del volcado en binario y el contenido en hexadecimal, de la Sección 2.3. Y por último se indica los parámetros de la variable expresado en 32 bits de caracteres que muestra el tamaño en bytes (04 00 00 00) del proceso, además sabiendo si el valor de la variable es válido (01) o no, de la Sección 2.4. Como existen cambios o sucesos en el programa ya sea de detención forzosa o reinicio del mismo, la variable se incrementa según los sucesos presentados, cambiando así el valor de tiempo de ejecución (TIMEOUT) de los procesos como se ilustra en la Figura 7.

```
[desarrollador@freebsd1 /tmp]$ sudo hexdump -C hMPRA7Ukp.rec
00000000  43 52 41 53 48 2d 52 45  43 4f 56 45 52 59 30 31  |CRASH-RECOVERY01|
00000010  46 41 49 4c 55 52 45 5f  44 45 54 45 43 54 4f 52  |FAILURE_DETECTOR|
00000020  54 49 4d 45 4f 55 54 00  00 00 00 00 00 00 00 00  |TIMEOUT.........|
00000030  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000040  04 00 00 00 00 00 00 00  01 ff ff ff ff ff ff ff  |................|
00000050  01 00 00 00 54 49 4d 45  4f 55 54 00 00 00 00 00  |....TIMEOUT.....|
00000060  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000070  00 00 00 00 04 00 00 00  00 00 00 00 01 ff ff ff  |................|
00000080  ff ff ff ff 02 00 00 00  54 49 4d 45 4f 55 54 00  |........TIMEOUT.|
00000090  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000000a0  00 00 00 00 00 00 00 00  04 00 00 00 00 00 00 00  |................|
```

Figura 7. Vista hexadecimal de una de las variables en disco.

Para el método de comprobación que carga estos valores se gestiona por el módulo una vez que se reintegra el proceso, donde el método loadRecoveryData de la Sección 2.4, busca el valor del nombre y tamaño en bytes de la variable como se ilustra en la Figura 8. Esta comprobación permite verificar el número de veces que la variable tuvo problemas o fallos, depende del tiempo esperado de recuperación (TIMEOUT), para que cada proceso pueda entrar en ejecución.

```
[desarrollador@freebsd1 ~/codeblocks/df]$ sudo bin/Debug/df --interface vtnet0
[CRM] CrashRecoveryManager(FAILURE_DETECTOR)
[CRM] tryOpen() found /tmp/nTdhSlrOr.rec
[CRM] loadRecoveryData() loaded TIMEOUT = 1
[CRM] loadRecoveryData() loaded TIMEOUT = 2
[CRM] loadRecoveryData() loaded TIMEOUT = 3
[CRM] loadRecoveryData() loaded TIMEOUT = 4
[CRM] loadRecoveryData() loaded TIMEOUT = 5
[CRM] loadRecoveryData() loaded TIMEOUT = 6
[CRM] loadRecoveryData() loaded TIMEOUT = 7
[CRM] recoverVariable(TIMEOUT) = 7
```

Figura 8. Funcionamiento del programa FDDemo con variable de recuperación.

El módulo de recuperación busca una variable (.rec) candidata a recuperar en caso de fallos, para cada variable que sea válida, depende si se está ejecutando varias instancias del mismo proceso en el mismo equipo; se tiene cuidado con la variable de recuperación ya que ésta es registrada más de una vez. Por otro lado, si un proceso se detiene no puede ser recuperado por otra instancia que no le pertenezca a su estado original, todo esto gracias al control de concurrencia synchronize.

La funcionalidad básica del módulo dentro de un programa (FDDemo) es marcar las variables de recuperación como estables, con el objetivo de que cada vez que se cambien o modifiquen esas variables sus valores puedan sincronizarse con la variable en disco para que si el proceso falla, la siguiente vez que se recupere tenga los mismos valores que se ejecutaron originalmente.

4. CONCLUSIONES

La creación de un módulo de recuperación de fallos de los procesos dentro de un sistema operativo FreeBSD, da la pauta para expandir a trabajos futuros en la implementación de un módulo compatible con diferentes plataformas, permitiendo así la interoperabilidad segura entre los sistemas.

El sistema o programa busca mantenerse en ejecución para lo cual se tiene un respaldo de los valores con los que estaba trabajando en proceso, una vez que se detiene o falla y desea reintegrase en un entorno distribuido de red, el módulo entra en acción permitiendo así poder realizar la recuperación del proceso con su estado original.

Cada variable o fichero que se crea en disco tiene la confiabilidad de que será recuperado y gestionado sincrónicamente por el proceso específico en ejecución. Sincronizar cada proceso con la variable en disco, depende parcialmente de los valores que sean proporcionados por el módulo en procesamiento en tiempo real.

La finalidad que se presenta en esta investigación es crear un módulo de recuperación de fallos, para que un sistema o programa sea capaz de gestionar errores de ejecución de cada proceso que tenga comunicación entre instancias de red en un sistema distribuido de equipos.

AGRADECIMIENTOS

Agradecimientos: Al Consorcio Ecuatoriano para el Desarrollo de Internet Avanzado CEDIA, por el financiamiento brindado a la investigación, desarrollo e innovación, mediante los proyectos CEPRA, en especial al proyecto CEPRA-X-2016-01; Sistema de comunicación fiable con caídas y recuperaciones de equipos con una seguridad mediante identificación anónima.

REFERENCIAS

Abts, D., Roberts, M., & Lilja, D. J. (2000). *A balanced approach to high-level verification: Performance trade-offs in verifying large-scale multiprocessors*. Paper presented at the Proceedings of the International Conference on Parallel Processing, January 2000, 505-510.

Angela, V., Granizo, R., Alex, A., & Tacuri, U. (2016). *Guía referencial para el manejo de QoS en redes WLAN prioritizando tráfico*. MASKANA: Actas CEDIA 2016, 65-77.

Chowdhury, T., & Mustafa, R. (2010). *Parallel data transmission: A proposed multi-layered reference model*. Paper presented at the Novel Algorithms and Techniques in Telecommunications and Networking, 139-142.

Claus, C., Zeppenfeld, J., Müller, F., & Stechele, W. (2007). *Using partial-run-time reconfigurable hardware to accelerate video processing in driver assistance system*. Paper presented at the Proceedings -Design, Automation and Test in Europe, DATE, 498-503.

Dickens, P. M., & Thakur, R. (1999). *Improving collective I/O performance using threads. In Parallel Processing, 1999. 13th International and 10th Symposium on Parallel and Distributed Processing*, Proceedings (pp. 38-45). IEEE.

Kamoshida, Y., & Taura, K. (2008). *Scalable data gathering for real-time monitoring systems on distributed computing*. Paper presented at the Proceedings CCGRID 2008 - 8th IEEE International Symposium on Cluster Computing and the Grid, 425-432.

Kleiman, S., Shah, D., & Smaalders, B. (1996). *Programming with threads* (p. 48). Sun Soft Press.

Jung, I., Hyun, J., Lee, J., & Ma, J. (2001). Two-phase barrier: A synchronization primitive for improving the processor utilization. *International Journal of Parallel Programming, 29*(6), 607-627.

McKusick, M. K., Neville-Neil, G. V., & Watson, R. N. (2014). *The design and implementation of the FreeBSD operating system*. Pearson Education.

Merz, F., Falke, S., & Sinz, C. (2012). *LLBMC: Bounded model checking of C and C++ programs using a compiler IR. In International Conference on Verified Software: Tools, Theories, Experiments* (pp. 146-161). Springer Berlin Heidelberg.

Mitterbauer, J., Zeilinger, H., Kohlhauser, S., Turek, T., & Kreilmeier, M. (2011). *Reliable data distribution in voice communication environments: Performance analysis of the group communication systems corosync and spread*. Paper presented at the IEEE AFRICON Conference.

Pina, L., & Hicks, M. (2016). *Tedsuto: A general framework for testing dynamic software updates*. Paper presented at the Proceedings - 2016 IEEE International Conference on Software Testing, Verification and Validation, ICST 2016, 278-288.

Plow, G. M. (1983). *Method and means for cataloging data sets using dual keyed data sets and direct pointers*. U.S. Patent No 4,408,273.

Spread Toolkit (2017). Accessed from http://www.spread.org/.

Sutter, H. (2005). A fundamental turn toward concurrency in software. *Dr.Dobb's Journal, 30*(3), 16-22.

Trott, O., & Olson, A. J. (2010). AutoDock Vina: improving the speed and accuracy of docking with a new scoring function, efficient optimization, and multithreading. *Journal of Computational Chemistry, 31*(2), 455-461.