Universidad de la República, Facultad de Ingeniería

Migración de procesos entre computadoras en un ambiente controlado

Sebastián Ventura

Tutor:
Ariel Sabiguero

Tribunal:
Javier Baliosian
Lorena Etcheverry
Franco Robledo

14 de diciembre de 2011

Resumen

Este trabajo ofrece una extensa explicación de la migración de procesos al mismo tiempo que se describen sus principales funcionalidades y objetivos. Se explican los sistemas que tienen este propósito y los problemas que estos tienen cuando se debe llegar a la práctica, y como podrían ser mejorados con un sistema orientado al usuario. Además, se demuestra cómo se puede mejorar el aprovechamiento de los recursos en el futuro, para poder ofrecer una distribución de procesos balanceada entre todos los sistemas que estén conectados a una red.

Las implementaciones de migraciones de procesos que existen actualmente fallan en no enfocarse en ser fáciles de usar y simples, por lo que difícilmente han visto aplicaciones prácticas en situaciones comunes. Los principales actores en los sistemas de migraciones de procesos son instituciones científicas con grandes computaciones que necesitan aprovechar todos los recursos, pero en este trabajo se apunta a un sistema en el que todos puedan aprovechar los recursos existentes, sobre todo teniendo en cuenta la cantidad de procesamiento que se ofrece con los sistemas móviles y con las computadoras personales.

De esta manera, se presenta el sistema procs, un sistema de migración de procesos orientado al usuario que ofrece distribución de procesos en la red y seguridad en la ejecución de los mismos. Como base para este programa se toman ideas de trabajos anteriores similares, pero que tienen problemas de usabilidad o que se encuentran desactualizados.

Se concluye el trabajo ideando la posible extensión del prototipo para ofrecer mejores opciones de migración y una distribución a través de una red `peer-to-peer`. Se demuestra así que esto no es solo posible, sino que es una de las funcionalidades que nos plantea el futuro cuando se nota la cantidad de recursos desaprovechados y el crecimiento del poder de las computadoras. Así como una vez las mainframes fueron reemplazadas por computadoras que lograban hacer localmente los cálculos, la migración de procesos reemplazaría a la nube como una distribución igualitaria y sistemática de los recursos que se encuentran en Internet.

Índice

1. Introducción

La principal misión de este trabajo es poder brindar una mirada global al problema de migración de procesos para luego poder adentrarse en la creación de un prototipo de migración en circunstancias controladas y con un enfoque al usuario. Es decir, diseñar un sistema de migración de procesos de propósito general que sea fácil de usar por el usuario, implementado una parte del mismo como demostración del poder que pueden tener las migraciones de procesos.

Es así que en esta, la primera sección, se realiza una introducción al tema, brindando los conceptos teóricos necesarios y las ideas que hay detrás de la mayoría de los sistemas de migración de procesos. También se presentan las principales motivaciones del trabajo y los objetivos que se tienen al plantear los problemas. En la segunda sección se hace una revisión de la migración de procesos y se explican sus funcionalidades y sus principales características; además también se introducen varias taxonomías para clasificar a los distintos sistemas existentes. En la tercera parte de este trabajo se van a presentar los principales sistemas que han existido para resolver este problema y se realiza una pequeña conclusión del estado actual de los sistemas para poder rever y explicar los objetivos del problema.

Luego en la cuarta sección, se diseña un sistema capaz de realizar una migración de procesos a nivel de usuario de forma simple y directa, como presentación de una solución a un problema más grande del que se va a implementar. De esta manera, en la quinta parte del trabajo se presenta la implementación realizada y las funcionalidades del prototipo, así como también sus principales casos de uso y su funcionamiento.

Por último, en la sexta sección se realizan conclusiones del trabajo y del estado actual, presentando la posibilidad de mejorar el trabajo existente y los usos que se le pueden dar a los sistemas de migración de procesos. Es necesario leer y entender las secciones primera, tercera y quinta para lograr llegar a las conclusiones que se plantean. Es así que este trabajo pretende brindar una solución global a partir de sistemas anteriores a un problema teórico muy investigado, pero que necesita más aplicaciones y soluciones prácticas.

1.1. Definiciones técnicas

Para poder definir la `migración de procesos`, es preciso primero evaluar la definición misma de un proceso en el sentido de la computación, y a partir de esta poder entender cómo se realiza una migración. Un `proceso` [1] es una instancia de un programa que está siendo ejecutado y que por lo tanto, contiene toda la información necesaria para que el programa funcione correctamente. De esta manera, se entiende un proceso como una unidad funcional que contiene el código fuente del programa y sus diferentes datos (que pueden llegar a ser el stack, variables, etc.). Mientras se ha avanzado en el desarrollo de los sistemas operativos, se han agregado a un proceso distintas cualidades para permitir la interacción entre este y el ambiente, dentro de las cuales las más prominentes son los archivos abiertos referenciados por el programa y los sockets o puertos abiertos que el programa utiliza (que en el caso especial del kernel Linux, tiene una definición parecida a los archivos abiertos).

Dada esta definición, se puede ahora a partir de la definición de migración, sobreentender la definición del problema que se está intentando investigar. Es

decir, si se entiende `migración` como el concepto de mover un objeto entre un lugar y otro manteniendo su funcionalidad, la migración de procesos se entendería como: la acción de mover un proceso desde un procesador hacia otro manteniendo su ejecución. Por lo tanto, al avanzar sobre esta definición, se entiende la migración de procesos [1] entre computadoras, como la transmisión de la unidad de procesamiento de una computadora a otra, permitiendo la ejecución de esta misma unidad en la segunda computadora. Pensando en programas, se entendería esta definición como continuar la ejecución de un programa en una computadora distinta de la cual se encuentra actualmente.

También es necesario para poder seguir discutiendo el tema, poder entender la definición de un `sistema distribuido` [2]. Este tipo de sistemas se define como el conjunto de múltiples computadoras autónomas que interactúan a través de una red de computadoras para poder conseguir una meta común. En otras palabras, se entiende a un sistema distribuido como las interconexiones necesarias que se dan entre distintos sistemas autónomos para de esta manera cumplir con un objetivo establecido. En cuanto a la migración de procesos, si esta se realiza en un sistema distribuido, significa que los procesos necesarios para poder realizar la migración se ejecutan entre varias computadoras interconectadas entre sí. También es importante notar que la migración de procesos en un sistema no distribuido existe, como se explica posteriormente, y se realiza a través de computadoras especializadas para la ejecución de los procesos.

Por otro lado, acoplado a la noción de sistemas distribuidos, se tiene la definición de sistemas `peer-to-peer` [3], que son aquellos que se comunican entre pares, es decir entre computadoras que tiene capacidades similares. De esta manera, se elimina totalmente la necesidad de centralización y se debe implementar un sistema de descubrimiento de pares. La anonimicidad que brindan estos sistemas y sus posibilidades por no tener un único punto de falla, ha logrado que se tenga mucha adopción de estos sistemas para la transferencia de archivos, aunque también han sido ampliamente estudiados por aprovechar de forma efectiva la red en la que se encuentran.

1.2. Migración de procesos

Dada la definición anterior, para ahondar más en el tema de la migración, se intentará brindar una simple idea sobre las cualidades que acompañan a la migración de procesos y sus distintas propiedades que ayudan a que sea una solución utilizada para distintos problemas. Primero, es imprescindible entender que actualmente existen varias soluciones para realizar las migraciones y que cada una ofrece sus propias cualidades. En esta sección se intentará brindar una mirada general a las opciones que se brindan a través de la utilización de la migración de procesos.

Se entiende que una migración de procesos es `transparente` [4] cuando esta se realiza sin que el proceso que está siendo migrado tenga conocimiento del cambio de sistemas al cual se encuentra sometido. De esta manera, se permite que un proceso no necesite estar especializado para lograr que la migración sea exitosa. Esta propiedad que algunos sistemas brindan tiene sus beneficios, entre ellos, que el programa no necesite ser codificado con la migración en mente y que el procesamiento no se vea complicado por nuevas variables del entorno que no son controladas. Por el otro lado, la transparencia obliga al sistema que brinda la migración a complicar su implementación, debido a que debe simular todo el

entorno anterior a la migración que tenía el proceso (archivos abiertos, sockets, etc.).

También es importante notar que no todas las soluciones a la migración de procesos brindan las funcionalidades necesarias para que el proceso se ejecute sin dificultades. Por ejemplo, es posible realizar una migración de procesos que decida no mantener los archivos abiertos y que delegue este problema al proceso (manteniendo igual la transparencia) indicando un error que el archivo fue cerrado o no existe. De esta manera también se quita dificultad al sistema y se aprovechan las funcionalidades que deben tener los procesos para poder recuperarse de errores imprevistos.

Las migraciones de procesos, por lo tanto, realizan un servicio de proveer al sistema de una computadora con la funcionalidad de poder reubicar procesos. Los beneficios de realizar esta migración son variables, dependientes en el proceso y en el ambiente, pero pueden ser comprobados en ciertas situaciones y se notan mayormente en procesos que tengan duraciones largas. El sistema desde el cual se migra el proceso se ve recompensado con una menor utilización de sus recursos que, por lo tanto, pueden ser reutilizados por otros procesos de mayor prioridad o que sean más locales (es decir, que sean más relativos al sistema en sí).

1.3. Virtualización

Un método utilizado para la migración de procesos es la virtualización de los mismos, para separarlos de las llamadas al sistema operativo y poder guardar el estado del proceso sin comunicarse con funciones de bajo nivel; esto se puede ver en las referencias [5, 6]. Para entender el uso de la virtualización es necesario entender su definición y funcionamiento.

La `virtualización` en general significa la utilización de software para crear una versión no real de un objeto. Es decir, un sistema que interactúa con un proceso de la misma manera que si fuera una versión real de un objeto, pero realizando distinto funcionamiento interno. En el caso de la virtualización para la migración de procesos, es interesante observar la virtualización de las llamadas al sistema que realiza un proceso; crear un sistema que actúe de intermediario entre el proceso y el sistema operativo, haciendo creer al proceso que se comunica directo con el último.

Al crear una virtualización, se puede lograr que el sistema guarde el estado del proceso y que pueda enviar señales como si se tratara del sistema operativo mismo, haciendo que el proceso permanezca sin modificaciones para poder ser migrado. Esto significa que la virtualización, al recibir las llamadas del proceso, además de redirigirlas al sistema operativo o al sistema que sea necesario, guarda información para poder conocer el estado del proceso; de la misma manera, el sistema envía señales al proceso y se comunica con él para poder cambiar su estado y migrarlo.

1.4. Balanceo de carga

Una de las motivaciones recurrentes de la implementación de sistemas basados en migración de procesos es el `balanceo de carga` [7], que en este contexto se entiende como la distribución de procesos para poder lograr la utilización

óptima de los recursos brindados por las distintas computadoras. En sí, el balance de carga permite que computadoras interconectadas puedan aprovechar al máximo los recursos existentes en el total de su conjunto. Así se permite que al interconectar computadoras se obtenga un mejor rendimiento que al tener las computadoras separadas.

El balanceo de carga se puede ver principalmente en sistemas más chicos al observar sistemas multiprocesadores (SMP), ya que el scheduler lo utiliza para poder decidir en qué CPU debe ser ejecutado cada proceso. El balanceo de carga en la migración de procesos es entonces una extensión de esta misma planificación a varias computadoras, de tal manera de permitir que cada proceso obtenga su mejor tiempo de ejecución (teniendo en cuenta prioridades si es necesario).

1.5. Clusters y sistemas de imagen única

Los `clusters` [8] son conjuntos de computadoras interconectadas que trabajan con un propósito, tratando de hacerse ver como un único recurso computacional para ciertas necesidades. Se utilizan sobre todo para el balance de carga y la tolerancia a fallas. Por lo tanto, son uno de los objetivos más importantes en la mayoría de las soluciones a la migración de procesos.

Los `sistemas de imagen única` [9] son un tipo de cluster donde realmente se presenta un conjunto de computadoras al resto de la red como un único recurso computacional en todos los aspectos. Están ampliamente relacionados con los sistemas operativos distribuidos (aquellos que están creados para ser utilizados en varias computadoras al mismo tiempo). Este es uno de los ejemplos de clusters que están más relacionados con la migración de procesos, ya que en estos sistemas es necesario tener un balanceo de carga dinámica y un movimiento de procesos.

1.6. Procesos en GNU/Linux

Para este trabajo y para entender distintos sistemas de migración de procesos, es necesario entender la representación de procesos en GNU/Linux [10] y sus distintas características. De esta manera, en esta sección se detallan las propiedades más importantes de los procesos en este sistema operativo para que cuando se entre en más detalle en la implementación de los sistemas se entienda su funcionamiento.

Identificación. Los procesos en GNU/Linux son identificados por un número llamado PID, que es único para cada proceso del sistema. A diferencia del estándar POSIX, GNU/Linux brinda PID para cada thread de un proceso, por lo que técnicamente, cada hilo de un programa puede ser estudiado para obtener su memoria tanto local como compartida. Además, las utilidades que tiene este sistema operativo permiten obtener el identificador de un proceso a partir del nombre del ejecutable, aunque esta conversión puede no ser única debido a que más de un ejecutable puede tener cierto nombre.

Memoria. Para poder ejecutar, un proceso hace uso de la memoria para guardar tanto toda la información necesaria para ejecutar correctamente, como los datos del ejecutable mismo. Es así que, si se puede leer toda la memoria que

un proceso tiene alocada (es decir, pedida para sí mismo), se puede obtener el código del programa compilado; es decir: lo que se está ejecutando actualmente, todos los datos de variables estáticas y dinámicas, y hasta las direcciones de retorno de llamadas a funciones o saltos.

En GNU/Linux es fácil distinguir por lo menos cinco áreas en la memoria de un proceso: la sección text, que contiene las instrucciones del programa o ejecutable en cuestión (es decir el código de máquina que el proceso ejecuta); la sección .data y la sección .bss, que contienen respectivamente las variables estáticas inicializadas con cierto valor y las no inicializadas (que usualmente se entienden como inicializadas en 0); el heap, que contiene la memoria dinámica que el programa ha requerido, por lo que incluye las variables dinámicas; y el stack, que contiene todas las palabras de memoria que se han puesto en el stack así como también las direcciones de retorno de funciones y de saltos.

En este sistema operativo, las secciones de memoria se organizan con un sistema llamado `VMA (Virtual Memory Areas)`, que mapea las secciones de memoria con las que un proceso trabaja, a direcciones de memoria virtual real de la computadora. Estas áreas de memoria sirven tanto para la seguridad de los procesos y que de esta manera solo puedan acceder a las partes de memoria que el sistema operativo les brinda (y así no pueden ver la memoria de otros procesos), como para que los procesos simplifiquen el acceso a la memoria y el guardar las distintas direcciones a las que apuntan las variables. Además, cada área de memoria permite tener distintos permisos como para poder ser compartida, o para saber si puede ser modificada, leída o ejecutada.

Es importante para que una migración de procesos funcione correctamente que todas estas secciones de memorias sean restauradas correctamente con todas las palabras que contenía el proceso original.

Bibliotecas dinámicas. En GNU/Linux, como en otros sistemas operativos, se tienen bibliotecas dinámicas. Se definen como conjuntos de funciones de código ejecutable que pueden ser cargadas en un programa posteriormente al comienzo de la ejecución. Además, tienen la ventaja de que la memoria virtual en donde se encuentran, puede ser compartida por distintos procesos ya que las funciones son la mismas. De esta manera, toda la memoria de una biblioteca (como es su sección text, su sección data y bss) es cargada en el VMA del proceso.

Cuando la biblioteca dinámica es cargada en el VMA, simplemente se agregan todas las áreas que sean necesarias para mapear las secciones de la biblioteca en la memoria local del proceso y que de esta manera ésta pueda referenciarla. Así es como el proceso, cuando utiliza alguna de las funciones que la biblioteca provee, hace una llamada en el código a la dirección de la VMA que contiene el código de la biblioteca.

Descriptores de archivos. Los descriptores de archivos abiertos son una forma que provee el sistema operativo para tener interacción entre un proceso y su sistema de archivos. Cada proceso que utiliza uno de los archivos que el sistema operativo le permite, genera un identificador único en el proceso que sirve para referenciar el archivo y la posición en la cual el proceso está leyendo y/o escribiendo en el mismo.

Además de los archivos que el proceso abre explícitamente, GNU/Linux también le permite a un proceso interactuar con tres descriptores de archivos específicos por defecto que son: la entrada estándar, la salida estándar y el error estándar; que tienen respectivamente, si no se modifica la ejecución normal, los identificadores 1, 2 y 3. Como según la filosofía UNIX casi todos los recursos son tomados como archivos por el sistema operativo y por sus procesos, esto facilita la interacción de un proceso con la línea de comandos o con archivos de log, y la redirección de la salida, de la entrada y/o del error sin que el proceso tenga que sufrir modificaciones. Es así que GNU/Linux implementa los llamados pipes del sistema que sirven para redirigir cualquiera de estos descriptores estándar hacia otro proceso.

Registros. Un poco más abajo en el nivel de abstracción, cada proceso además debe guardar en su estado el valor de cada uno de los registros locales de la arquitectura que utiliza el proceso. Esto se debe a que cuando ocurre un cambio de contexto, el proceso que debe ser restaurado debe poder guardar nuevamente en cada uno de los registros los valores que espera tener para poder ejecutar las operaciones de la arquitectura, y además para poder indicar el estado de ejecución del proceso, como se da con los registros que apuntan a la instrucción siguiente del proceso y al stack del mismo.

Esto significa que una migración de procesos debe poder guardar todos estos registros y poder restaurarlos como si fuera un cambio de contexto local a la máquina. Solo de esta manera el proceso puede seguir realizando las instrucciones que le brinda la arquitectura y mantener el estado del mismo correctamente para indicar la instrucción que se ejecuta y la dirección de memoria en donde se encuentra el stack del mismo. Todos estos registros son dependientes de la arquitectura, por lo que se explica un poco más su utilización en la sección de arquitectura (5.1.4).

1.7. Motivación

La principal motivación de este desarrollo es el estudio de la actual situación de los sistemas de migración de procesos para poder inspeccionar la factibilidad de la implementación de sistemas distribuidos de balanceo de carga de procesos para el uso personal. En otras palabras, se intentará brindar una visión global del estado de los sistemas que utilizan migración de procesos para luego poder investigar la implementación de un nuevo sistema que brinde la migración de procesos en un uso cotidiano.

Este problema existe debido a que hoy en día las soluciones existentes no proveen de una manera fácil la migración ni el balanceo de carga de tal manera que pueda ser utilizado por el `usuario común`. Es además importante mencionar que muchas de las soluciones existentes, como se explica posteriormente, carecen de ciertas propiedades o contienen ciertas limitaciones que serían inaceptables para una migración que esté basada en el usuario. Además, se verá como estas soluciones están pensadas con objetivos distintos a los que se pretende llegar con este desarrollo y que por lo tanto, no logran proveer una solución para un rango de problemas importantes.

Se puede además notar la necesidad de una solución de este estilo gracias al gran crecimiento que han tenido hoy en día las computadoras personales y los `dispositivos móviles` que contienen en sí mismo un CPU con capacidades

cada vez mayores. Es de esta manera que se nota que en cualquier lugar, hoy en día, hay muchos recursos computacionales que no están siendo utilizados y que por lo tanto pueden ser considerados como desperdiciados, mientras que otros aparatos tienen necesidad de recursos que no pueden ser cumplidas por distintas razones (por ejemplo, en celulares no se puede tener procesadores con mucho poder debido a limitaciones de espacio y batería, similar a lo que sucede con las computadoras portátiles). Un ejemplo clásico es el uso de un computador personal pequeño (como por ejemplo las netbooks) que no tiene mucha capacidad de procesamiento, mientras una computadora de escritorio no es utilizada y por lo tanto desperdicia sus recursos.

La diferencia entre la utilización de las computadoras actualmente y como se usaban hace una década explica muy bien la necesidad de tener estas soluciones. No es extraño ver hoy que el trabajo que antes se realizaba en servidores (rendering, compilación, etc.) se esté realizando cada vez más en las `estaciones de trabajo` ya que tienen mayor capacidad. Esto hace que las computadoras sean adquiridas y diseñadas para su uso máximo, ya que un uso medio de las mismas solo significa actualmente la utilización de un explorador para acceder a Internet. Por lo tanto, la mayoría del tiempo, estos recursos se ven inutilizados mientras puede haber otras estaciones de trabajo que los necesiten.

Además, el poder computacional que una persona tiene hoy en día distribuido en sus distintos aparatos electrónicos, es comparable con lo que las instituciones científicas tenían hace dos décadas, dando a entender que estos recursos podrían ser aprovechados para otras aplicaciones. Se notará como hay varios ejemplos de utilizaciones de este tipo a través de migración de procesos y balanceo de carga, pero que la mayoría de las opciones son solo aprovechadas por instituciones científicas.

De esta manera, en este estudio se pretende brindar soluciones a la falta de recursos en aparatos personales siempre que se puedan encontrar recursos en otras computadoras, y siempre que la ejecución de procesos se vea beneficiada por este cambio. Se podrá ver en el resto del estudio cómo las distintas ideas e implementaciones que existen pueden ser aprovechadas y mejoradas para brindar una experiencia más adaptada al usuario y que permita la utilización común de un sistema de migración de procesos.

2. Migración de procesos

En esta sección se realiza una mirada global al problema de migración de procesos y se explica el funcionamiento general de estos sistemas. De esta manera, se pretende poder brindar una introducción más técnica al problema que permita entender los problemas y las soluciones más comunes para la transferencia de un programa en ejecución de una computadora a otra.

Se plantean primero los objetivos de los sistemas de migración para conocer los problemas que son resueltos por estos y de esta manera explicar las situaciones en las que se utilizan y con qué propósito. Se explican posteriormente las características principales y los requisitos de los sistemas, y para poder luego explicar con más detalle un algoritmo genérico de migración de procesos. Esta sección se concluye con las distintas taxonomías del sistema y por último las aplicaciones prácticas que se les da a los sistemas.

2.1. Objetivos

Los sistemas de migración de procesos tienen como objetivo facilitar la ejecución remota de programas para evitar el desuso de recursos. De esta manera, son utilizados para solucionar problemas de distintos tipos, y aunque los objetivos específicos dependen de cada aplicación, algunas de las metas [11, 12] principales de la utilización de migración de procesos se explican a continuación.

Uno de los objetivos más importantes de la migración de procesos es la `distribución de recursos`, pues así puede aprovechar de manera eficaz el poder computacional que se tiene. Esta idea es especialmente utilizada en investigaciones científicas donde se requieren grandes tiempos de computación y se tienen distintas opciones para poder utilizar procesadores que de otra manera serían inalcanzables. De esta manera se logra tener un mejor rendimiento global que se basa en las eficiencias locales.

Relacionado a este tema está el `balanceo de carga`, que justamente permite distribuir procesos para aprovechar los recursos. Poder realizar un eficaz balanceo es importante tanto para aplicaciones que necesitan respuestas rápidas como para las que requieren de mucho poder computacional. La migración de procesos es importante en clusters, por ejemplo (aunque no sea preemptivo), donde un servidor recibe pedidos para ejecutar procesos y según la carga que tenga cada uno de los otros servidores se redistribuyen los procesos para de esta manera aprovechar todos los recursos disponibles.

Como se explicaba anteriormente, con el crecimiento de las computadoras portátiles se ha visto también un crecimiento en la necesidad de poder migrar procesos entre los distintos aparatos tecnológicos que se pueda tener, y es por eso que la `computación móvil` es un objetivo a futuro. Es importante tener en cuenta este objetivo para entender la importancia actual de la migración de procesos, ya que los recursos disponibles en computadoras portables son cada vez mayores y pueden ser mayormente aprovechados con soluciones de migración de procesos adecuados.

Se tiene en cuenta para la migración la `localidad de los recursos` para poder migrar procesos que son iniciados en una procesador que no es el que tiene el mejor alcance a los recursos necesarios para obtener la mejor eficiencia. Esto se puede ver por ejemplo en dos servidores, cada uno con datos en disco distintos, donde los procesos necesitan distintos archivos y podrían ser migrados entre los

servidores dependiendo en dónde se encuentre el archivo necesario. También es importante tener en cuenta el caso en que los recursos son inaccesibles desde otra computadora y es obligatorio migrar el proceso para poder acceder a los mismos.

`La tolerancia a fallas` se puede obtener con migración de procesos donde se implemente una solución que permita que, cuando haya una falla en la computadora o procesador que esta ejecutando el proceso, se migre hacia otro donde pueda ejecutar sin problemas. De esta manera se establece un camino por el cual los procesos no se vean restringidos por los problemas de cada una de las computadoras. Además, esto también brinda posibilidades en `la administración de las computadoras` gracias a la oportunidad de tener servicios con gran disponibilidad, debido a que pueden ser migrados cuando se haga mantenimiento sobre una de las computadoras.

2.2. Características

Los algoritmos para poder lograr la migración de procesos comparten ciertas características que tienen importancia como para notar en este estudio. En esta sección se presentan algunas de las más importantes para poder luego adentrarse en el algoritmo específico que permite la migración [1].

Como se explicaba anteriormente, se deben entender los algoritmos de migración de procesos como `algoritmos distribuidos`. Es decir, que dadas dos computadoras que participan en una migración de procesos, cada una está corriendo una parte del algoritmo para poder efectuar la migración correctamente. Por ejemplo, el caso más simple es un algoritmo que maneje una parte emisora y otra receptora.

Como es distribuido y es muy dependiente de las llamadas al sistema operativo, los sistemas de migración de procesos son `algoritmos complejos` que contienen código poco portable. Esto se nota más en especifico en la sección 2.3.

Para la correcta funcionalidad de estos sistemas, es necesario que la migración de procesos se realice con gran `comunicación con el sistema operativo` donde se ejecuta el sistema. Esto se debe sobre todo a la posibilidad de iniciar y restaurar procesos como será explicado posteriormente.

De esta manera, al contener todas estas características, claramente el sistema debe contener dentro de sí mismo el `estado` actual de sus propios procesos, así como también es posible que contenga el estado de los procesos asociados a la migración. También es posible que el sistema guarde información sobre otros nodos que participan en el algoritmo de migración de procesos.

Más allá de esta corta explicación de las características del algoritmo, es necesario ver cómo todas juegan un papel importante en el diseño de un sistema de migración. Es así que se debe ver un algoritmo de migración como un problema distribuido, complejo, altamente acoplado con el sistema operativo y que sea capaz de guardar y entender su estado, pues esta es la única manera que permite que la migración se realice correctamente y con el acuerdo de todas las partes.

2.3. Requisitos

Para que se pueda realizar una migración de procesos existen, como se explica en la sección 2.5, dos posibilidades en cuanto a los permisos que necesita el

sistema. Cuando el sistema está implementado para ejecutarse como una aplicación de usuarios, necesita tener un soporte del sistema operativo en cuanto a la interacción con los procesos, lo cual lo diferencia de la otra opción ya que aquella permite interactuar directamente con los objetos del sistema operativo. Por lo tanto, se plantean aquí los requisitos [11, 1] que debe soportar el sistema operativo para que una aplicación en el espacio de usuario pueda realizar una migración de procesos, excluyendo aquellos básicos de un sistema operativo moderno (como por ejemplo, poder utilizar redes de computadoras).

Lo primero y más importante es justamente la `exportación e importación de procesos`, pues es la que permite guardar un estado de un proceso y restaurarlo. El sistema operativo debe brindar alguna manera por la cual un proceso pueda acceder a la información de otros procesos para exportarla e importarla. Es decir, deben existir llamadas al sistema operativo con las cuales se pueda obtener el estado actual del proceso y otras llamadas por las cuales se pueda establecer el estado de un proceso.

Para poder guardar un estado es necesario que el sistema tenga `acceso a la información del proceso y sus recursos.` El sistema de migración de procesos debe tener acceso a cierta información del proceso, como por ejemplo su nombre, para poder recrear un contexto igual en el ambiente objetivo, además de también poder obtener toda la información sobre archivos abiertos u otros recursos que el proceso esté utilizando si van a ser migrados. Solo de esta manera, el sistema puede brindar total transparencia de migración, recreando en el sistema migrado todo el contexto para que el proceso migrado se ejecute sin problemas.

Cuando se quiere terminar una migración, luego de acceder y guardar el estado de un proceso, es necesario poder `dejar de ejecutar procesos y limpiar residuos de la migración.` Así se deja el sistema en el cual el proceso ejecuta antes de la migración sin cambios. Es necesario que el sistema operativo brinde una interfaz que permita a la migración de procesos remover del mismo toda la información del proceso una vez migrado. Por ejemplo, esto puede significar tener la posibilidad de matar un proceso, para que de esta manera se deje de ejecutar y elimine la información suya de las tablas del sistema operativo.

Por último, tomando en cuenta el objetivo de balanceo de carga, es un requisito del sistema poder acceder a la `información del sistema` y así poder comprobar si es necesario realizar una migración. Si se quiere utilizar la migración de procesos como un sistema de balanceo de carga, es necesario que el sistema operativo provea información sobre su propio estado para que de esta manera el sistema implementado para la migración pueda decidir cuándo es necesario migrar y cuál va a ser el ambiente objetivo.

2.4. Algoritmo genérico

Aunque hay distintas variaciones para realizar la migración de un proceso, a continuación se explicará una breve y concisa descripción de cada uno de los pasos necesarios de la migración. Es importante notar que en algunos sistemas los pasos pueden estar intercambiados o pueden ser no existentes dependiendo de la implementación.

1. Si se realiza balanceo de carga, calcular y seleccionar proceso para migrar.
2. Si se realiza balanceo de carga, elegir sistema al cual migrar el proceso.

3. Enviar un pedido al sistema objetivo para negociar la migración.
4. Se suspende el proceso que se va a migrar.
5. Se empieza a encolar toda comunicación hacia el proceso para reenviarla posteriormente al proceso migrado (o se descarta enviando el mensaje de error correspondiente).
6. Se extrae el estado del proceso a migrar y se envía al sistema objetivo.
7. Se crea un nuevo proceso en el sistema objetivo que va a ser el destino.
8. Se transfiere el estado recibido al proceso destino y se reanuda la ejecución del mismo.
9. Se mantiene un reenvío de comunicación desde el sistema original al destino, indefinida o temporalmente, y se envían las comunicaciones encoladas.
10. Se elimina el proceso en el sistema original y todos sus residuos.

Este algoritmo genérico permite dar a entender los pasos que se realizan en una migración. Aunque son pasos simples, cada uno contiene dentro de sí una complejidad mayor y una cantidad de sub etapas que se deben llevar a cabo para la transferencia de un proceso. A modo de ejemplo, la extracción del estado de un proceso es una etapa que debe realizar varias llamadas al sistema operativo para poder completar sus estructuras internas y de esta manera poder enviarlas.

Es importante notar que aunque aquí se plantee el algoritmo como una lista de pasos, varios de estos pueden estar ejecutando todo el tiempo en una de las máquinas (como calcular el estado del sistema) o pueden ser ejecutados paralelamente o de forma entrelazada (como obtener el estado de un proceso e irlo enviando a medida que se tenga información). Además, como es un algoritmo distribuido, varios de estos pasos se pueden realizar en distintas computadoras (por ejemplo, quien calcula el estado puede ser distinto del sistema que envía el proceso).

2.5. Variaciones y taxonomías

Se presentan a continuación distintos tipos de taxonomías que se aplican a los sistemas de migración de procesos que fueron tomadas de las referencias [13, 1, 14, 7, 6], con cierta modificación en alguna de las partes para que quede más completo y más específico a los sistemas de migración de procesos. Posteriormente se clasificarán los distintos ejemplos, así como también el prototipo aquí presentado según ellas.

2.5.1. Según dónde se ejecuta el sistema

Primero que nada, una de las características más importantes de cualquier sistema es si el proceso ejecuta como una aplicación de usuario o de kernel. Es decir, se diferencian los sistemas en cuanto a su relación con el sistema operativo y con el proceso que va a ser migrado. De esta manera, los sistemas pueden pertenecer a más de una de estas clases porque su relación con el sistema y con el proceso pueden ser distintas. Se explican a continuación cinco posibilidades para la ejecución un sistema de migración.

- `En el espacio de direcciones del kernel.` El sistema se implementa usualmente como un módulo que puede ser cargado en el kernel y que se ejecuta en el espacio de este para obtener más fácilmente la información del proceso. En general un sistema implementado de esta manera tiene una mejor eficiencia de los recursos a costa de modificar el kernel sobre el cual corren los procesos. Esta orientación se ve implementada en varios de los sistemas que se ejecutan sobre UNIX y derivados.

- `En el espacio de direcciones de usuario.` Estos sistemas aprovechan las llamadas del sistema que provee el sistema operativo para obtener la información necesaria del proceso, sin interferir en el proceso mismo. Aunque tienen menor eficiencia, son transparentes y más fáciles de adaptar e instalar.

- `Microkernel.` De forma parecida a la clase anterior, pero con la diferencia de que las llamadas al sistema se realizan de manera distinta debido a la diferencia del sistema operativo. También se maneja en base a llamadas del sistema, pero en aquellas que le permiten comunicarse con otros procesos que corren sobre el espacio del usuario.

- `En el espacio de direcciones de la aplicación.` A diferencia de las clases anteriores, esta requiere modificación del código del proceso a ser migrado ya que se basa en sí mismo para realizar la migración. La transparencia y la reusabilidad son sacrificadas para obtener un mejor rendimiento basado en características específicas del proceso a ser migrado.

- `Virtualización.` Aunque en realidad el sistema se ejecuta sobre el espacio de direcciones del usuario, es importante separar la opción de virtualización por la diferencia que esta puede tener respecto a otras soluciones. Estas implementaciones actúan como intermediario en la comunicación entre el proceso y el sistema operativo para recopilar información y crear, matar y reanudar procesos.

2.5.2. Según cómo se realiza la transferencia de procesos

Como los sistemas de migración de procesos envían los procesos hacia otras computadoras, es importante distinguirlos en cuanto a cómo realizan esa transferencia. Aunque todos utilizan de una manera u otra la red para que el proceso llegue a otro sistema, se nota que la forma en que se utiliza es importante para entender cómo funcionan los sistemas. Un sistema puede implementar las dos clases de migraciones, pero esto sucede poco en la práctica. Las posibilidades son:

- `Archivo.` En este modo se utilizan archivos compartidos donde se guardan los datos del procesos, como por ejemplo en un NFS (Network File System) [15].

- `Red.` A diferencia del anterior, los datos se envían directamente a través de la red sin necesitar archivos compartidos.

2.5.3. Según cómo se realiza la transferencia de datos

Se explican a continuación las diferentes posibilidades que existen en cuanto a cuándo se transfieren los datos necesarios para que el sistema funcione. Esta categorización es usualmente aplicada a la copia de procesos, ya sea en un mismo sistema o para la migración de procesos. Para la copia local de los procesos, esta taxonomía describe usualmente la copia de memoria, pero para la migración de procesos, como se necesita casi toda la memoria para poder ejecutar el proceso remotamente, generalmente sirve más para explicar la transferencia de datos extras (como por ejemplo, archivos).

- `Eager (all).` Se copia previamente a la migración del proceso, toda la información respecto a este y todos los datos relacionados.
- `Eager (dirty).` Si el sistema operativo provee de paginación de memoria con bit dirty, este algoritmo solo envía aquellas que han sido modificadas, mientras que las otras se proveen a demanda desde otro medio de almacenamiento.
- `Copy-On-Reference.` Solo se transfieren las páginas desde memoria cuando se hace referencia a ellas.
- `Copy-On-Write.` Solo se transfieren las páginas desde memoria cuando se escribe a ellas, mientras tanto se mantienen compartidas.
- `Flushing.` Realiza la escritura a disco de las páginas modificadas, y luego se transmiten desde disco cuando se hace referencia a las mismas.
- `Precopy.` Se transfieren las páginas modificadas sin dejar de ejecutar el proceso en el origen para que, cuando se ejecute el proceso en el destino, ya se encuentren en el sistema. Si se modifican páginas mientras se realiza el precopy, se deben enviar otra vez.

2.5.4. Según el balanceo de carga

El balanceo de carga en un sistema de migración de procesos es similar al balanceo de carga general, y es por eso que se plantean aquí las tres categorías más generales para realizar el balanceo de carga. Aunque al parecer disjuntas, estas opciones se superponen pues se pueden realizar distintos balanceos según el proceso; además la última categoría contiene a las otras dos, por lo que se puede ver un balanceo de carga estático preemptivo, es decir que migre procesos en ejecución según reglas específicas y sin tomar en cuenta la carga del sistema. Las tres clases de balanceo de carga son:

- `Estático.` Los procesos en ejecución se asignan a sistemas sin tener en consideración la carga actual del sistema.
- `Dinámico.` Los procesos en ejecución se asignan a sistemas según la carga actual en cada uno de ellos.
- `Preemptivo.` Los procesos se migran dinámicamente en tiempo de ejecución.

2.5.5. Para balanceo de carga, según cómo se decide el destino

Cuando se realiza un balanceo de carga para una migración de procesos, se debe tener un algoritmo que permita elegir el sistema al cual se le enviará el proceso. Según este algoritmo, se toman decisiones distintas que pueden afectar el funcionamiento del balanceo, por lo que es importante distinguir estas cinco posibilidades para poder entender la elección del sistema destino. Se plantean a continuación estas opciones, que pueden estar entrelazadas, ya que un algoritmo puede ser implementado con lo mejor de cada una de ellas.

- `Política iniciada por emisor.` El sistema que quiere realizar una migración envía pedidos hacia otros sistemas para realizarlo.
- `Política iniciada por receptor.` El sistema que puede recibir una migración envía periódicamente información sobre su estado.
- `Políticas simétricas.` Tanto el emisor como el receptor pueden enviar pedidos para realizar la migración.
- `Políticas azarosas.` El destino se elige al azar por el sistema emisor.
- `Árbitro.` Existe un nodo de la red que mantiene información de los nodos y es quien decide el sistema destino.

2.6. Aplicaciones

La categorización de los sistemas sirve para describir de manera general el funcionamiento de los sistemas y poder distinguir los programas existentes de migración de procesos. Sin embargo es tan o más importante entender cómo se pueden aplicar los conceptos y los beneficios planteados por estos sistemas en la práctica. Es así que a continuación se explica la orientación de los programas al usuario y se concluye con ejemplos reales de utilización de este tipo de herramientas.

2.6.1. Tipos de aplicaciones beneficiadas

Es oportuno para este caso desarrollar una nueva taxonomía para la migración de procesos en base al usuario final para el que está pensado el sistema. Esto sirve para entender el vacío actual que existe en la migración de procesos orientada al usuario. Es por eso que se presentan a continuación clases tentativas para clasificar a los distintos sistemas:

- `Aplicaciones científicas y empresariales.` Estas soluciones usualmente están implementadas para investigaciones científicas o para cálculos empresariales que necesiten de muchos recursos para poder ser ejecutados. Aunque existe una gran cantidad de estos sistemas, lo que sucede es que al estar enfocadas a institutos y empresas poseen muchas opciones de configuración y una dificultad de aprendizaje que no pueden existir en aplicaciones para usuarios corrientes. Más adelante se explicarán detalladamente algunas de sus fortalezas y debilidades para cada uno de los sistemas que fueron implementados de esta manera.

- `Aplicaciones para usuarios.` Los sistemas de esta categoría son usualmente pequeños programas que permiten a un usuario con suficientes permisos realizar una migración de un proceso actualmente en ejecución de manera simple. Sin embargo, usualmente requieren que el usuario siempre interactúe con el programa (a veces hasta realizando el usuario la transferencia del proceso exportado) y no proveen de soluciones para balanceo de carga. En esta categoría se basa este actual desarrollo.

- `Pruebas de concepto.` Dentro de la migración de procesos también existen sistemas que fueron implementados con la única idea de comprobar desarrollos científicos y que por lo tanto, son casi imposibles de configurar y poco portables. Estas soluciones permiten investigar y poder comprobar el rendimiento de las migraciones de procesos pero son de poca utilidad en el uso real de una aplicación.

2.6.2. Ejemplos generales

De forma parecida a la categorización de los objetivos, los sistemas de migración de procesos pueden ser usados con varios propósitos e ideas. Se brindará una mirada general a las aplicaciones que utilizan migración y de qué manera aprovechan sus beneficios, para de esta manera poder concebir el beneficio que puede proveer un sistema de migración a los distintos programas existentes.

Las aplicaciones que de manera más obvia y clara son beneficiadas por la migración de procesos, son aquellas que han sido programadas para tener en cuenta sus distribución y migración. Es así como en la siguiente sección se explican varias aplicaciones que permiten la creación de estas aplicaciones, y que permiten una distribución más eficaz de los recursos que un simple balanceo de carga. Sin embargo, como se explica a continuación, existen otros programas que se pueden ver beneficiados por los sistemas de migración de procesos sin estar pensados para ello.

Por ejemplo, los procesos que tienen `larga ejecución`, como en aplicaciones científicas, son de los más utilizados hoy en día para aprovechar la migración de procesos. Se pueden adaptar para poder ser guardados y migrados, para de esta manera evitar pérdida de información y tener tolerancia a fallas. Además, se puede lograr con un buen balanceo de carga preemptivo que las aplicaciones ejecuten en menos tiempo debido a un aprovechamiento más eficaz de los recursos.

Si tomamos en cuenta que muchas de las aplicaciones pueden ser `distribuidas`, se puede notar como ejecutar distintas partes del programa en distintos nodos, siempre aprovechando la localidad de los datos y una utilización eficaz de los recursos, beneficia a los programas para que puedan ejecutar de manera más eficiente y más rápida. Si intercalamos esta noción con la anterior, se ve cómo las aplicaciones científicas de larga duración que han sido distribuidas pueden aprovechar al máximo las posibilidades que se brindan gracias a los sistemas de migración de procesos.

Por otro lado, los programas que reciben pedidos en aplicaciones `multiusuario` se podrían ver beneficiadas con la migración al poder acercarse más hacia los recursos necesarios para los distintos requerimientos, o incluso para poder atender de manera más eficiente los pedidos que distintos usuarios requieren.

Como última posibilidad, y una que requiere más estudio, se encuentra la migración de `aplicaciones móviles.` Una aplicación que se ejecuta sobre una computadora móvil está ajustada para disponer de menos recursos debido a la escasez de estos en la computación móvil. Sin embargo, con un ambiente de migración de procesos no solo se podría generar un mejor rendimiento, sino que también se podría crear una mejor interacción con el usuario.

De esta manera, se detalló cómo la migración de procesos que ha sido explicada en este capítulo puede proveer grandes beneficios a aplicaciones reales y que merece un estudio profundo para poder proveer una solución general y que brinde soluciones a varios problemas que existen hoy en día, especialmente enfocándose en la mejora del aprovechamiento de los recursos y la mejor eficiencia en la ejecución de programas.

3. Trabajos relacionados

Se presentarán a continuación distintos trabajos relacionados con el tema, intersectándolos con el desarrollo de este proyecto. Se intentará dar una visión global del estado en el que se encuentra la migración de procesos en sistemas distribuidos que permitirá posteriormente adentrarse en la explicación del prototipo a implementar. Hay muchos trabajos sobre este tema, pero se tratará de ser conciso y exponer solo aquellos que son más relevantes sin ser demasiado impreciso.

3.1. Sistemas

Para la explicación de los distintos sistemas se seccionará cada uno en tres partes: objetivos, diseño y funcionamiento. En la primera se describe cuáles son los problemas que el sistema intenta resolver y cuáles son las arquitecturas y sistemas operativos para los cuales fue diseñado. En la segunda parte se plantea breve descripción del diseño del sistema con sus distintas partes. Y en la tercer y última sección se hará un breve resumen sobre su funcionamiento y rendimiento. Se presentan entonces estos trabajos, ordenados por el año en que aparecieron y por la categoría general a la que pertenecen.

3.1.1. Primeros pasos

XOS (X-TREE Operating System) [16, 1, 12] es un sistema operativo basado en la arquitectura X-TREE y, por lo tanto, a diferencia de otros ejemplos, es creado para el estudio del diseño de sistemas operativos sobre esta arquitectura. La arquitectura X-TREE es un sistema multiprocesador simétrico donde los procesadores se ordenan en un árbol para comunicarse. La principal misión de este sistema operativo era proveer de migración de procesos para poder acercarlos a los recursos necesarios en la arquitectura X-TREE

Se implementa un sistema de mensajes que facilita la migración de procesos, ya que es la vía principal de comunicación del sistema operativo. Parecido a otros sistemas operativos distribuidos, esto permite que los procesos puedan ser totalmente manejados a través de los canales de comunicación establecidos en la arquitectura X-TREE.

El sistema operativo en sí no es muy utilizado, aunque la arquitectura X-TREE ha visto sus implementaciones funcionales. De todas maneras, al ser un sistema a nivel de kernel, es poco portable y es considerado más bien como una prueba de concepto para demostrar las capacidades de esta arquitectura.

DEMOS/MP [17, 1, 12] es otro ejemplo de una implementación de un sistema operativo distribuido cuya principal misión es el balanceo de carga. De esta manera, se le agregó un sistema de migración de procesos para tener una balanceo preemptivo, y por lo tanto un mejor aprovechamiento de los recursos.

Se provee de un sistema de comunicación e intercambio de mensajes del cual el kernel participa, gracias al cual se pudo implementar una migración de procesos totalmente transparente y eficiente. Además, al estar el sistema basado en estos mensajes, cuando se realiza un checkpoint se encolan todos los mensajes del proceso, permitiendo reenviarlos cuando el proceso se reanude, dejándolo en el estado correcto.

Siendo uno de los primeros ejemplos de este tipo de migración de procesos, fue utilizado brevemente y ha demostrado su eficacia en cuanto a la migración de procesos. Sin embargo, al ser un kernel totalmente diferente y no totalmente compatible con UNIX, no puede ser reutilizado hoy en día.

PVM (Parallel Virtual Machine) [18, 4, 14] es un sistema para paralelizar trabajos entre muchas computadoras, es decir, que sus distintos procesos se distribuyan sobre las estaciones de trabajo. Originalmente pensado para poder realizar cálculos paralelizables y muy usado para la enseñanza. Para poder ser utilizado en ambientes más complicados y estar a la par con los anteriores, se ha creado una versión de PVM llamada MPVM (Migratable Parallel virtual Machine) que incluye la migración de los procesos paralelizables. Tanto PVM como MPVM (y otras extensiones como DynamicPVM) tienen como misión principal agrupar un conjunto de recursos como un único recurso de procesamiento.

Hacer uso del sistema que provee PVM es simple, pero requiere cambios en el programa, ya que se deben usar bibliotecas especificas y definir lo que se llaman tareas. Cada tarea es un proceso del programa que se puede enviar hacia otros nodos del sistema para ser procesado. Cuando un programa es comenzado, el nodo local busca en un conjunto de nodos que conoce a cuál se le pueden enviar procesos, y de esta manera inicia la comunicación para negociar la migración. Las reglas para encontrar el nodo son variables y configurables.

Dentro del diseño, es importante mencionar que PVM es totalmente heterogéneo, es decir, que puede contener distintos recursos computacionales dentro de su red y seguir funcionando correctamente. Además, también provee un sistema para que las distintas tareas se comuniquen entre sí, y así no tener que hacer uso de otras formas más complicadas a través de la red.

MPVM es creado como una variación de PVM que provee migración transparente, en un intento de aprovechar todos los recursos computacionales disponibles mientras se permite el uso de las estaciones de trabajo para otras tareas cuando sea necesario. No utiliza checkpointing, sino que se transfiere todo el espacio virtual del proceso, lo que usualmente lleva a que sea más lento en la migración que otros programas. Utiliza un nodo central para dirigir la planificación.

La performance de MPVM es comparable con la de usar PVM sobre MOSIX, a diferencia de que MOSIX tiene más beneficios para el usuario. Sin embargo, al ejecutar en el espacio de usuario, es más simple de implementar y además permite que el sistema sea heterogéneo. Además, PVM puede permitir una mejor paralelización de los trabajos ya que no es transparente (es el programador quien decide cómo paralelizar los procesos y hasta puede definir en qué nodos ejecutar). Sin embargo, debido a su objetivo, la aplicación está orientada a instituciones científicas.

3.1.2. Migraciones Transparentes

LOCUS [19, 1, 12] es un sistema operativo distribuido compatible con UNIX, cuya idea fue crear un Single System Image para poder ejecutar varios procesos sobre un cluster de computadoras. Por lo tanto, es un sistema creado para balanceo de carga, lo que significa proveer de un sistema de archivos distribuidos y un planificador de procesos común entre los distintos nodos.

Una de las decisiones más importantes de diseño de este sistema es el sistema de archivos. Se reemplazaron las partes de UNIX para trabajar de manera distribuida y con procesos de replicación de archivos. De esta manera se aprovecha al máximo la ideología de UNIX, donde todo es un archivo y las interacciones se realizan con ellos. A partir de esto, LOCUS también provee de mecanismos limitados para comunicación entre procesos y de migración de los mismos preemptivamente.

LOCUS es uno de los pocos ejemplos de sistemas operativos que alcanzó el estado de estar listo para producción. Al ser una de las primeras implementaciones de este tipo, ha sido relativamente estudiado y mejorado para tener un mejor alcance. Siendo un sistema a nivel del kernel, provee transparencia de procesos y una implementación para transferencia de datos eager-dirty.

MOSIX [4, 1, 18] y sus derivados [7] fueron creados con el propósito de tener un sistema operativo distribuido para el aprovechamiento de clusters. MOSIX fue creado a partir de la versión 7 de UNIX con adiciones para proveer a los procesos de un sistema distribuido con migración de procesos preemptivo, y de esta manera poder configurar fácilmente una cantidad escalable de estaciones de trabajo como si fueran un cluster.

Sus principales objetivos y características son: migración de procesos preemptiva para el balanceo de carga, imagen única del sistema para unificar todas las estaciones de trabajo, autonomía en cada nodo del cluster, configuración dinámica (necesaria para un sistema en el espacio del kernel), control descentralizado y escalabilidad.

Todos los recursos en un dominio de sistemas MOSIX tienen un identificador único para que se pueda crear la imagen única del sistema y que se puedan acceder a todos los recursos disponibles. Para lograr la utilización de estos identificadores únicos se tiene un sistema por el cual cada recurso es identificado en su nodo y de esta manera poder ser referenciado. Esta tecnología es comparable a NFS.

MOSIX tiene un planificador agresivo, donde a diferencia de otros sistemas, se hace un balanceo de carga continuo para reagrupar los procesos en distintos nodos. Siendo totalmente distribuido, cada nodo guarda información de un conjunto al azar de nodos que son considerados para ser receptores de una migración de procesos. Sin embargo, en estos casos no provee tolerancia a fallas en cada nodo (sí globalmente), ya que los procesos que corren en un nodo fallido no son migrados nuevamente.

La performance de MOSIX ha sido medida reiteradas veces y se ha comprobado su utilidad, no solo por la teoría, sino también en la práctica. Como un sistema de migración de procesos, presenta las funcionalidades de ser preemptivo, transparente y en el espacio de direcciones de kernel; todas estas características y el hecho de que utilice una copia de datos eager-dirty lo vuelven una muy buena solución para instituciones científicas, pero de difícil adopción para el usuario corriente.

Sprite [1] es un sistema operativo con la misión de tratar una red de computadoras como una computadora de tiempo compartido, pero con la performance de un sistema de un solo usuario. Por lo tanto, es un sistema operativo dis-

tribuido con un sistema distribuido de archivos y con una imagen única del sistema.

Se diferencian los procesos entre los que ejecutan en su sistema local y los que terminan siendo migrados a otras computadoras. Se redirigen todas las llamadas del sistema necesarias al nodo que inició la ejecución del proceso. Además, tiene la posibilidad de iniciar directamente el proceso en otra máquina o de migrarlo preemptivamente.

Al ser todo un sistema operativo, provee de grandes posibilidades pero al mismo tiempo cerrándose a otras opciones. De todas maneras, es un buen ejemplo de la mejoría que se puede lograr con migración de procesos y tolerancia a fallas.

Mach [1] es un microkernel desarrollado por la Universidad de Carnegie Mellon. Un sistema de migración de procesos fue implementado sobre este sistema, utilizando virtualización, de tal manera de poder experimentar con migración de procesos sobre un microkernel.

El diseño del sistema de migración de procesos es más simple que otros ejemplos debido a que se basa en la distribución de memoria que ya realiza el microkernel y el sistema distribuido de comunicación de procesos. Se soporta una total transparencia en el nivel de usuario.

Al ser un sistema implementado para experimentar, simplemente sirve como una prueba de concepto, aunque posee varias propiedades que son importantes para ser estudiadas. Utiliza un sistema de copia de memoria por cada fallo de página para evitar una copia excesiva de información.

Inferno [20, 21] es un sistema operativo distribuido creado para ser comercializado a partir de Plan 9 de Bell Labs. Sus principales objetivos son que todos los recursos sean referenciados como archivos, que las redes sean tomadas como un solo espacio coherente de recursos computacionales y proveer de un protocolo estándar de comunicación para todos los procesos y recursos.

Para lograr estos cometidos, Inferno utiliza virtualización de los procesos para poder aprovechar al máximo los recursos que se brindan con este tipo de sistemas. De esta manera, se crea un lenguaje llamado Limbo, que es el que se compila a la máquina virtual del sistema operativo, lo que brinda muchas posibilidades a la hora de migrar procesos.

Inferno no tuvo éxito comercial y ha sido liberado como software libre. Sus objetivos siguen siendo considerados ya que es un proyecto que todavía se encuentra en mantenimiento. Como es a nivel de kernel, es poco utilizado en ambientes de producción y principalmente es pensado para aplicaciones de computación sobre la grid y en la nube.

3.1.3. Espacio del usuario

Condor. Los objetivos de Condor [22, 23, 24, 1] son simples y parecidos a lo que se intenta en este desarrollo: aprovechar el desuso de las estaciones de trabajo que se da en lugares empresariales o institutos científicos. De esta manera, el sistema está centrado en brindar una solución integral y profesional, manteniendo la ideología open source, para utilizar todos los recursos disponibles para realizar computaciones de alto rendimiento con períodos largos de computación (HTC).

Es por eso que dentro de sus bases fundamentales se encuentra la recuperación de fallas (por si los nodos dejan de estar disponibles para la computación y dejan de responder o por si algún nodo falla en la computación distribuida), el balanceo de carga (para lograr un buen aprovechamiento de los recursos) y la administración de políticas (para la seguridad de las empresas e instituciones).

Es importante notar que la existencia de Condor está basada en estudios previos sobre el estado de la computación distribuida para HTC, donde se describen perfectamente los problemas que se piensan resolver. La primer investigación que llevó a la creación de Condor es un estudio del porcentaje de computación que estaba siendo desaprovechada por las estaciones de trabajo. Los otros estudios refieren a otros programas que solucionan este problema, pero sin tener en mente procesos de larga duración, debido a la pérdida de soluciones intermedias de los procesos.

Para el balanceo de carga, Condor utiliza un servidor central que a través de pedidos regulares a las estaciones de trabajo obtiene su estado, de esta manera obteniendo la información necesaria para migrar los procesos. Asimismo, cada estación tiene su propio planificador para poder decidir el uso de sus recursos entre los distintos procesos. Como Condor se basa en la inutilización de las computadoras, cuando alguna de ellas pasa a estar en estado activo (es decir que el usuario las empieza a usar), inmediatamente todos los procesos vuelven hacia el servidor central y/o otras estaciones de trabajo.

La migración se realiza a través de un sistema de checkpointing y reinicio de procesos. Cuando se debe matar un proceso de una estación para migrarlo, se hace un checkpoint de su estado, efectivamente guardando todas las variables que estaba utilizando el proceso, incluyendo su código. Para reiniciarlo, simplemente se carga un nuevo proceso con esta información y se recomienza su ejecución.

Condor es una de las soluciones más usadas para aprovechar los recursos de las computadoras de manera distribuida. Es una solución altamente probada, con alrededor de 20 años de desarrollo que es usada en muchos proyectos de gran importancia. Su gran capacidad de administración y su enfoque en proveer recursos a computaciones HTC se ha demostrado como un estándar en varios círculos de la informática y la ciencia. Esto se debe en parte a que los distintos estudios sobre su performance han demostrado ampliamente su utilidad.

Provee migración de procesos dinámica (preemptiva en los casos de fallas) y en el espacio de direcciones usuario, por lo cual es de suma importancia para este desarrollo (aunque utiliza NFS para compartir los datos de los procesos). Sin embargo, como en la migración usa copia eager-all y su enfoque es en aplicaciones empresariales y científicas, no es fácil de utilizar y se podría ver como perteneciente a la clase de programas que una vez configurados es mejor no modificar.

Platform LSF (Load Sharing Facility) [25, 1, 26] es un sistema de manejo de trabajos y procesos distribuidos en computadoras conectadas por red que permite la creación de clusters y computación de alta performance. Su misión es crear un sistema de balanceo de carga para conjuntos grandes y heterogéneos de computadoras. Así se idea un programa que tiene escalabilidad, permite la ejecución de procesos en paralelo y distribuidos, transparente y con velocidad de respuesta rápida a cambios en los recursos.

Una de las principales consideraciones para el diseño es su habilidad de ejecutarse en varios sistemas operativos, permitiendo un uso heterogéneo de los recursos. Además, sumado a esto su capacidad para ejecutar en el espacio de usuario, le permite un dinamismo que no se ve en muchos otros sistemas. Utiliza además un nodo central para poder calcular la información de carga de los nodos en el sistema.

Aunque LSF está principalmente basado en una migración de procesos dinámica antes de ser ejecutados, también contiene un sistema para migración de procesos preemptivo parecido al provisto por Condor. Es semi-transparente ya que se necesita agregar una biblioteca externa al programa para poder obtener toda la información del proceso al hacer el checkpoint.

La principal ventaja de LSF es su soporte comercial y su heterogeneidad. Sin embargo, tiene desventajas en distintos aspectos, como por ejemplo en el requisito de que haya un sistema de archivos compartido. Al contrario que otros sistemas, una de las principales ideas de LSF es que para el balanceo de carga, se le da más importancia al posicionamiento inicial, dejando de lado la migración preemptiva. Por ser principalmente un sistema de balanceo de carga, su orientación es claramente hacia empresas y ejecución de batch.

LoadLeveler [27, 23] es un planificador de tareas de IBM que permite, como los otros, la distribución de procesos y el control de los mismos. Su objetivo fue crear una versión empresarial derivada de Condor para aprovechar las soluciones que tiene IBM.

El diseño es en sí parecido a Condor y por lo tanto también maneja para la migración de procesos una versión de checkpointing y reinicio de procesos. Provee también una API para interactuar con el sistema y de esta manera poder extenderla para necesidades específicas. Comparte con PVM la habilidad de fácilmente paralelizar trabajos para su ejecución en un ambiente distribuido.

Lo importante de esta solución es su enfoque al ámbito empresarial y su alta integración con otros productos IBM.

PANTS (PANTS Application Node Transparency System) [8, 28] es un sistema para multiprocesamiento distribuido transparente, que permite ejecutar programas de varios procesos en un cluster Beowulf sin que los programas sean escritos específicamente para el sistema. Por lo tanto, su objetivo es empresarial y científico para institutos con recursos bajos que necesitan poder computacional. Por ser una solución para clusters, claramente uno de sus principales objetivos es el balanceo de carga.

Lo más interesante de esta aplicación es su diseño, ya que al tener que ejecutarse sobre un cluster Beowulf, su misión principal es requerir muy pocos recursos y no utilizar demasiado la red, que es usualmente el cuello de botella para procesos que ejecutan sobre un cluster.

Utiliza multicast por dos vías (una para el líder y la otra para los nodos desocupados), por donde se realiza la comunicación necesaria para obtener la información sobre los nodos y realizar un balanceo de carga. Es importante notar entonces que se tiene un sistema centralizado, pero al utilizar multicast, se provee la posibilidad que se cambie el líder cuando este esté demasiado ocupado.

Utiliza dos aplicaciones a nivel de usuario, una para controlar el balanceo de carga y otra que es la que sirve para la ejecución remota de los procesos antes de

comenzar a ejecutarse (PREX). Para la migración de procesos, usa EPCKPT para hacer checkpointing y reiniciar los procesos preemptivamente.

Aunque PANTS es un muy buen ejemplo al nivel de diseño, al desarrollarse sobre un cluster, es necesario verlo como una solución a problemas científicos y empresariales. De todas maneras, su utilización efectiva de EPCKPT y la transparencia que permite son de gran valor para su estudio.

3.1.4. Bibliotecas y aplicaciones

Libcpkpt [29, 30, 14] fue creada para ser una biblioteca para UNIX de checkpointing transparente, principalmente para tolerancia a fallas, se ha visto utilizada en la migración de procesos debido a su utilidad para realizar checkpointing incrementales.

Lo más importante del diseño de esta biblioteca es que es un proceso de usuario, lo cual significa no necesitar ninguna modificación en el kernel. Sin embargo, para mantener la simplicidad de la implementación, se necesita modificar el código del programa que necesitará ser recuperado y ser re-linkeditado con la biblioteca.

Además, también es notable mencionar que el sistema de checkpointing que utiliza es avanzado, ya que no solo se realizan checkpoints incrementales regulares para ser tolerante a fallas, sino que también provee de un sistema de checkpointing dirigido por el usuario para excluir ciertas partes de la memoria o realizar un checkpointing sincrónico usando llamadas directamente en el código del programa.

Es una buena biblioteca de nivel de usuario que permite un checkpointing semi-transparente con copy-on-write, pero carece de soluciones para ciertas llamadas de paralelismo en el código. De tal manera, al igual que EPCKPT, pertenece a las pruebas de concepto, donde lo importante es poder estudiar su implementación y no tanto su uso.

EPCKPT (Eduardo Pinheiro Checkpoint Project) [30] es un proyecto personal en el que cual se intentó crear una modificación del kernel Linux para checkpointing que cubra ciertas debilidades de otros proyectos. Específicamente, los problemas que se nombran son: poder hacer checkpoint de cualquier aplicación, limitar y minimizar la imagen de un proceso y ser totalmente transparente.

Al ejecutarse en el espacio del kernel se tiene acceso a toda la información de los procesos por lo tanto su implementación es más simple que otros sistemas. Se agregan tres nuevas llamadas al sistema para que un proceso pueda lograr la migración.

Según el artículo de presentación de la aplicación, se tiene buen rendimiento de hasta un 23 % de mejora. Esta aplicación es un buen ejemplo de una implementación simple para migración de procesos que podría ser reutilizada en otros proyectos. Sin embargo, al estar desactualizada (por ejemplo, no hay versión para la línea 2.6 del kernel de Linux) solo resulta de uso para el estudio de su funcionamiento.

LinuxPMI (Linux Process Migration Infrastructure) [31] es una serie de patches para el sistema operativo GNU/Linux para habilitar la migración de procesos en el espacio del kernel. Por lo tanto, su principal misión es claramente

proveer con un método simple para migración de procesos. Fue creada a partir de openMOSIX para soportar las nuevas versiones de GNU/Linux.

Al ser un derivado de MOSIX, su diseño es similar en cuanto al uso de recursos. Además, se provee comunicación entre procesos a través de pipes de GNU/Linux. Sin embargo, se crearon varias restricciones sobre las características que deben tener los procesos para ser migrables.

El sistema se encuentra totalmente desarrollado y es continuamente actualizado. No ha tenido mucha adopción porque es un conjunto de patches al kernel de Linux, pero las características que comparte con MOSIX y su capacidad de ser un sistema distribuido, lo hacen un buen ejemplo para este desarrollo.

DMTCP (Distributed Multi-Threaded Checkpointing) [32] fue creado como una migración de procesos a nivel de aplicación que tiene soporte para distintos programas. Su principal utilización fue pensada para ámbitos científicos donde estos programas requieren grandes recursos computacionales.

Como está explicado en su sigla, está implementado para ser un sistema distribuido de checkpointing de procesos con varios hilos. De esta manera provee también de una API para la comunicación con el sistema.

Es un buen ejemplo de migración de procesos a nivel de aplicación y por lo tanto en el nivel de usuario, sin que sea necesaria ninguna modificación al kernel subyaciente, aunque actualmente solo esta soportado en el sistema operativo GNU/Linux. Por su diseño de solo reemplazar ciertas llamadas al sistema, provee de un buen tiempo de respuesta.

Ghost Process [6] es un servicio a nivel de kernel para el sistema operativo GNU/Linux, específicamente para su derivado Kerrighed. Tiene como objetivos proveer de virtualización para mayor facilidad en la creación de sistemas de única imagen y clusters.

Al utilizar virtualización, todos los datos del proceso pueden ser obtenidos por el proceso y si es necesario, redirigir los pedidos del proceso si este ha sido migrado. Así se logra una simplicidad en la implementación y una API que puede ser aprovechada por otras aplicaciones para proveer de migración de procesos. Utiliza checkpointing para la migración.

Aunque con una teoría fuerte, su implementación no pasa más allá de ser una prueba de concepto, por lo que no puede ser reutilizado para otras soluciones. Además, también necesita modificaciones del kernel, lo que hace que en sí el sistema sea complicado para ser un sistema basado en virtualización.

CryoPID [33] es un programa de nivel de usuario para poder fácilmente hacer checkpointing de un proceso y reiniciarlo. Su principal misión es demostrar la factibilidad de esta idea, y no tanto la migración de procesos. Es un sistema de tolerancia a fallas y que puede ser útil para guardar estados de programas.

Utiliza las distintas funcionalidades que provee GNU/Linux para guardar todo el estado del proceso, aunque ciertas propiedades no pueden ser guardadas actualmente, como algunos archivos abiertos, aplicaciones gráficas y sockets.

Se nota que es un buen ejemplo de un sistema que tiene funcionalidades a nivel de usuario y sin privilegios necesarios para poder suspender procesos y volverlos a iniciar. Sin embargo, como no es su principal misión la migración, no tiene grandes utilizaciones en esta área.

CosMiC [13] es un conjunto de bibliotecas orientadas a ofrecer checkpointing para la tolerancia de fallas y la migración de procesos. Entre sus principales objetivos se encuentran el total aprovechamiento de los recursos computacionales a través de balanceo de carga y proveer migración de procesos para tolerancia a fallas.

Se separa el sistema en cuatro bibliotecas distintas, cada una con sus casos de uso específicos. Existe tanto la migración de procesos transparentes como el checkpointing de datos críticos para evitar pérdida de datos. Tiene una estructura distribuida con varios procesos que se encargan de ejecutar remotamente y planificar los procesos.

Se creó específicamente para resolver problemas con largos tiempos de procesamiento y no ha sido mejorado para incluir comunicación entre procesos y otros tipos de mensajes. Está basada en sistemas operativos UNIX, por lo que su código es útil para entender el funcionamiento de las migraciones. Es importante notar que aunque es extensible, se encuentra desactualizado.

3.1.5. Aplicaciones recientes

BOINC (Berkeley Open Infrastructure for Network Computing) [34, 35] es un sistema computacional distribuido para aplicaciones de grandes tiempos computacionales. Su objetivo es crear un sistema simple para que todas las computadoras conectadas a Internet puedan compartir sus recursos inutilizados para que aplicaciones específicamente creadas para este propósito puedan ejecutarse más rápidamente.

BOINC es en sí un sistema de computación voluntaria que pretende proveer de un framework para el trabajo de otras aplicaciones que ya existían en esta área (como SETI@HOME [36], por ejemplo). De esta manera, es una mejora respecto a los ejemplos de computación voluntaria que existían anteriormente ya que permite que se utilicen todos los recursos disponibles en Internet para las aplicaciones que el dueño del poder computacional desee.

Provee una migración de procesos anterior a la ejecución, donde cada computadora que se hace voluntaria del proyecto recibe un ejecutable y envía los resultados de vuelta al sistema. Por lo tanto, es un sistema centralizado de control. Además, también tiene comunicación entre los procesos y formas de generar interfaces gráficas. Es en sí una API o biblioteca con la cual se tienen que implementar los programas para poder funcionar.

Además de todo esto, se quiere también proveer control de sus computadoras a los voluntarios, por lo que se brinda una interfaz gráfica de administración y varias formas de lograr que los usuarios se sientan partícipes de los desarrollos que se están llevando a cabo.

Es uno de los sistemas con mayor adopción a nivel de usuario, ya que permite fácilmente compartir los recursos de una computadora y elegir para qué aplicaciones se deben utilizar. Ha tenido diversos éxitos en el ámbito y gran adopción también por parte de la comunidad científica que necesita grandes recursos computacionales.

Al no ser transparente ni proveer de migración preemptiva, es solo un buen ejemplo de cómo lograr un buen uso de esta clase de sistemas. Lamentablemente, el hecho de que haya que programar directamente con su API y bibliotecas lo hace más difícil de usar en producción.

VirtualBox Teleporting [37] y VMWare Live Migration [38]. Aunque son dos soluciones separadas y distintas, son las más utilizadas respecto a migración de procesos por sus objetivos. La idea principal de estas soluciones es migrar máquinas virtuales sin tener que apagar las mismas; esto significa que se puede seguir trabajando en una máquina virtual aunque esta haya sido movida de nodo.

Son soluciones empresariales específicas para ciertos contextos, donde lo más importante es tener un tiempo de actividad alto para funcionalidades que lo necesiten, usualmente servidores web o aplicaciones importantes de usuario. Es así que las dos están basadas en el sistema de máquinas virtuales al que pertenece su compañía y no se plantean como soluciones generales.

El diseño de los dos ejemplos es en sí simple, ya que se debe guardar toda la información de una máquina virtual, lo que sería algo así como suspenderla. Como el proceso que las migra se ejecuta fuera de estas y altamente acoplado con el sistema de máquinas virtuales, se tiene un alto conocimiento sobre todos los procesos que se ejecutan y la funcionalidad de los mismos.

Además, al estar basado en virtualización, se puede mantener toda la información de los procesos, inclusive las conexiones de red, ya que se puede mantener el estado mismo del sistema operativo. Lamentablemente, por razones de tamaño de las máquinas virtuales, es necesario que el archivo que representa el disco duro de la misma esté compartido entre el sistema origen y el destino.

Altamente utilizado en ambientes empresariales y científicos, es un buen ejemplo de los usos de la virtualización y el poder que se puede lograr con la migración de procesos. Como esto significa trabajar con los sistemas de máquinas virtuales de cada uno, se tienen varias restricciones que no pueden ser mejoradas. En sí, como ejemplo de migración de procesos, no han sido muy estudiadas en éste ámbito debido a esta última razón.

3.2. Estudios relacionados sin aplicaciones concretas

Se han realizados varios estudios sobre la migración de procesos y el balance de carga transparente con diferentes conclusiones. Es fácil de todas maneras, dividir la mayoría de estos estudios en aquellos que realmente hicieron una generalización del tema y un estudio de las distintas opciones, y en los que simplemente fueron creados para demostrar el funcionamiento de los nuevos programas.

Entre los primeros [1, 12, 4], que son los que realmente importan en esta sección, se puede ver cómo la gran variedad de casos que existen han tenido `poca aplicación` comercial y por los usuarios. Se discuten además muchas razones por las que se da esta situación, entre ellas: la falta de casos de uso para la migración de procesos del usuario medio y del mercado comercial, falta de soporte especializado en la red para este tipo de funcionamiento, y factores sociológicos para entender lo que significa que se ejecuten procesos ajenos.

Es importante mencionar que hay pocos estudios recientes sobre este tema, y la mayoría data de la década de los '90, por lo que los problemas que existían en ese momento no necesariamente se han mantenido, ni tampoco las soluciones que se pretendían brindar. Hoy en día existen `más situaciones` para la migración de procesos, ya que cada casa es un cluster electrónico con poderes computacionales mucho mayores que hace diez años. También la red ha mejorado mucho, hasta más rápido que las velocidades de acceso a memoria y a disco, lo que indica

una posibilidad de aprovechar mejor los recursos. Y por último, proyectos como BOINC han demostrado que es posible utilizar recursos ajenos sin que se tenga que entender qué es lo que se ejecuta.

Dentro de los estudios que sí son recientes y relevantes existen dos que llaman la atención en cuanto a su innovación para virtualización [39] y migración de procesos [40]. En el primer caso, se intenta proveer de virtualización al sistema operativo para móviles Android, lo cual significaría nuevas posibilidades para migración de procesos en esos sistemas. El segundo, trata un sistema innovativo que permitiría utilizar una cámara para sacar una foto y de esta manera reconocer la aplicación para ejecutarla en otra computadora; aunque no es realmente migración de procesos, se obtiene un sistema parecido que permite que las aplicaciones se migren desde un sistema a otro.

Más allá de las conclusiones, una idea interesante que aparece en distintos estudios son los WORMS, que estarían definidos como programas auto migrables que toman decisiones en base a su contexto de hacia adónde y cuándo migrar. Además, el programa debe tener en cuenta su estructura distribuida y programarse para tener en cuenta posibles fallos. Aunque no de mucho uso, es un buen ejercicio mental para entender los usos que se pueden llegar a tener gracias a la migración de procesos.

Por último, aquellos estudios que pretenden presentar nuevas tecnologías, vienen acompañados de resultados posiblemente cuestionables en cuanto a su rendimiento. De todas maneras, se nota que se pueden obtener mejores resultados con la migración de procesos; sobre todo con balanceo de carga dinámico, aunque también con migraciones preemptivas. Uno de los problemas que siempre se tiene en cuenta es el `tiempo` que toma realizar una migración, lo que hace que sea poco factible para procesos de corta duración.

3.3. Conclusión

Una comparación de las distintas tecnologías existentes se nota en el Cuadro 1. La columna transparencia en esta tabla indica si la migración de procesos es transparente, aunque se ha decidido que una migración soportada directamente por el sistema operativo no cuenta como totalmente transparente, ya que esta cualidad solo se debe al soporte del sistema operativo y no a consideraciones específicas de diseño. También se nota que MACH tiene implementación en el sistema operativo y en el nivel de usuario, ya que justamente su diseño es una aplicación de usuario altamente soportada por el microkernel. Por último, se nota que las bibliotecas no tienen balanceo de carga y que su migración es solamente preemptiva, ya que no toman en cuenta la carga de los sistemas; como además todos los otros sistemas tiene balanceo dinámico y preemptivo, con excepción de BOINC, se decidió no realizar un comparación en esa área.

Analizando los resultados de estudiar cada una de las posibilidades distintas, y tomando en cuenta que su orden es parecido al histórico, se pueden notar varios resultados importantes que influyen tanto en el orden que se debe realizar un estudio como en el diseño del prototipo. Es importante notar de todas maneras que estas características pueden llegar a ser muy genéricas para describir en profundidad cada uno de los sistemas, y es por eso que muchos resultan parecidos entre sí cuando técnicamente son muy distintos.

Primero que nada, se nota la evolución de los sistemas en el tiempo. Se ve que, aunque los estudios se empezaron a realizar en el nivel del kernel y

Sistema	Transparencia	Implementación	Transferencia de Procesos
XOS	Casi total	Sistema Operativo	Red
DEMOS/MP	Casi total	Sistema Operativo	Red
PVM/MPVM	Limitada	Nivel de usuario	Archivo
LOCUS	Casi total	Sistema Operativo	Archivo
MOSIX	Casi total	Sistema Operativo	Archivo
Sprite	Limitada	Sistema Operativo	Red
MACH	Casi total	Sistema Operativo/Nivel de usuario	Red
Inferno	Casi total	Sistema Operativo	Archivo
Condor	Biblioteca	Nivel de usuario	Archivo
LSF	Biblioteca	Nivel de usuario	Archivo
LoadLeveler	Biblioteca	Nivel de usuario	Archivo
PANTS	Total	Nivel de usuario	Archivo
Libcpkpt	Total	Nivel de usuario	Archivo
EPCKPT	Total	Nivel de usuario	Archivo
LinuxPMI	Casi total	Nivel de kernel	Archivo
DMTCP	Limitada	Aplicación	Archivo
Ghost Process	Limitada	Nivel de usuario	Archivo
CryoPID	Casi total	Nivel de usuario	Archivo
CosMiC	Casi total	Nivel de usuario	Archivo
BOINC	Limitada	Nivel de usuario	Archivo
VM migration	Total	-	Archivo

Sistema	Transferencia de Datos	Distribución	Aplicación
XOS	Eager(all)	Distribuido	Prueba de concepto
DEMOS/MP	Eager(all)	Distribuido	Científica
PVM/MPVM	Eager(all)	Distribuido	Científica
LOCUS	Eager(dirty)	Distribuido	Científica y empresarial
MOSIX	Eager(dirty)	Distribuido	Científica y empresarial
Sprite	Flushing	Centralizado	Prueba de concepto
MACH	Copy-On-Reference	Distribuido	Prueba de concepto
Inferno	Eager(all)	Distribuido	Científico y empresarial
Condor	Eager(all)	Centralizado	Científico y empresarial
LSF	Eager(all)	Centralizado	Científico y empresarial
LoadLeveler	Eager(all)	Centralizado	Científico y empresarial
PANTS	Eager(all)	Semi-Distribuido	Científico y empresarial
Libcpkpt	Eager(all)	-	Prueba de concepto
EPCKPT	Eager(all)	-	Científico
LinuxPMI	Eager(dirty)	Distribuido	Usuario
DMTCP	Eager(all)	Distribuido	Usuario
Ghost Process	Eager(all)	-	Usuario
CryoPID	Eager(all)	-	Usuario
CosMiC	Eager(all)	-	Usuario
BOINC	Eager(all)	Centralizado	Usuario
VM migration	Eager(all)/Precopy	-	Usuario

Cuadro 1: Comparación de Tecnologías

el sistema operativo, la mayoría de las últimas tecnologías intentan realizar la migración de procesos a `nivel de usuario`. Esto se debe a que en las décadas de los '80 y los '90, los sistemas operativos no estaban totalmente consolidados, por lo que muchas de las soluciones eran más fácilmente creadas a partir de un nuevo sistema en vez de soportar muchos distintos ambientes. Sin embargo, hoy en día hay cuatro sistemas operativos que son los estándares para este tipo de aplicaciones (Mac OS, GNU/Linux, Windows y BSD), de los cuales tres están basados en UNIX.

Además del avance hacia soluciones que se ejecuten en el nivel de usuario, también se ve un intento de encontrar como principal objetivo al usuario y una forma fácil de realizar la migración. Esto se puede deducir por el gran trabajo que ha habido en la comunidad académica para encontrar soluciones empresariales para clusters que sirven tanto a empresas como a instituciones científicas, lo que crea un vacío de soluciones de este tipo para el usuario corriente.

Por otro lado, también se nota cómo la mayoría de las soluciones comparten ciertas propiedades de diseño, principalmente por su `facilidad de aplicación`. Entre ellas, las más importantes son que las transferencias de procesos se realizan con archivos y las de datos con toda la memoria referenciada por el programa. En sí, estas dos soluciones brindan buena tolerancia a fallas, pero al mismo tiempo agregando más tiempo de migración.

En cuanto a la transparencia, como en la distribución del sistema, se nota que se han probado casi todas las opciones, desde poca transparencia hasta una separación total del proceso, pasando por usar bibliotecas intermedias. Asimismo, se han optado por crear tanto sistemas distribuidos como centralizados, sobre todo en la solución a problemas en clusters de computadoras.

Las investigaciones realizadas sobre este tema han provisto una buena visión en los problemas y en las soluciones que se han encontrado. Es importante enfocarse en los cambios que ha habido en el contexto computacional (sobre todo en el paralelismo de las aplicaciones para aprovechar los multiprocesadores) y en que las soluciones hasta el momento han tenido poco adopción global.

Desde toda esta visión histórica es que parte este proyecto, queriendo poder brindar una solución que se asemeje a las `aplicaciones comunes` de hoy en día, y que pueda también proveer soluciones a problemas de distribución. Por lo tanto, este desarrollo pretende brindar una idea de cómo se puede construir una aplicación que sea distribuida y dinámica, tratando de mantener las posibilidades que brinda la migración de procesos.

Es importante ver que este tema tiene una gran cantidad de `interés académico`, ya que aprovechar los recursos computacionales de forma total y eficiente es uno de los puntos más importantes de la distribución de procesos y del balanceo de carga. De esta manera, los cambios recientes en el poder computacional que se tiene en un ambiente corriente, han cambiado el paradigma para los programados, pasando de la pregunta "¿cómo hago que el programa se ejecute con bajos recursos?" a "¿cómo puedo aprovechar al máximo los recursos que tengo?". Por lo tanto, más que nunca, la migración de procesos y el balanceo de carga se están transformando en una necesidad debido a la gran cantidad de oferta de recursos que hay y la poca demanda.

La creciente cantidad de agentes móviles ha sido poco estudiada, tanto en estas tecnologías como en las investigaciones. Por lo tanto, también es importante tenerlas en cuenta para próximos estudios, ya que su cantidad y disponibilidad brindan posibilidades que no se han tenido anteriormente. Así, la solución que

se pretende brindar debe tener la característica de ser lo más portable posible para las nuevos sistemas operativos móviles que están apareciendo.

En conclusión, este tema ha sido estudiado profundamente con pocos resultados efectivos. Condor, BOINC y los derivados de MOSIX son casi los únicos ejemplos de tecnologías de este estilo que se utilizan hoy en día. Sin embargo, su ámbito de aplicación se ve restringido a los `institutos científicos`, sobre todo para aprovechar los beneficios.

Con todas las opciones existentes, se intentará obtener lo mejor de cada una para así lograr una solución que pueda ser utilizada por el usuario común y por las computadores en un hogar. Cada vez más se nota que los hogares tienen los poderes computacionales que las instituciones científicas tenían hace veinte años, pero por el distinto uso que se le da, `no se aprovecha todo su potencial` ni pueden ser acomodados a tecnologías viejas.

Se espera que este estudio pueda haber brindado una visión global del contexto para poder a partir de aquí, tratar el problema más profundamente. Para así encontrar la solución que se puede llegar a necesitar dentro de pocos años, para no malgastar los recursos y para lograr que los hogares puedan ser más inteligentes y más rápidos; al mismo tiempo brindando una opción para aquellos que necesitan mucho poder computacional.

4. Diseño del prototipo

Para la creación del programa se decidió crear un prototipo vertical, para así tener una base para todas las características básicas que debería tener un programa de este estilo. Se decidió en contra de usar las bibliotecas existentes (aunque quizá sí su código) o de mejorar algunos de los productos existentes debido a que sería más difícil por la diferencia en objetivos con la mayoría de ellos.

El diseño del prototipo está basado en tres principios: mantenerlo simple, amigable al usuario y extensible; tener en cuenta siempre la eficacia y realizar el mínimo de comunicación posible; y, por último, hacerlo lo más portable posible. Bajo estos principios se razonaron los requisitos mínimos que se deben cumplir y se tratará de implementar solo lo necesario para que se cumplan los tres objetivos, al mismo tiempo que los requisitos más básicos.

A continuación se presenta el diseño realizado para el prototipo para tener una idea del alcance del programa. Posteriormente se explicará su implementación y su uso.

4.1. Requisitos funcionales

Para empezar a entender el diseño del programa, se deben plantear los requisitos funcionales que el programa debe cumplir para poder posicionarse como una solución de migración de procesos con las ideas planteadas anteriormente. De esta manera, a continuación se detallan los requisitos mínimos que se consideran necesarios para que un sistema pueda funcionar correctamente en un ambiente distribuido con migración de procesos. Se eligen particularmente las funcionalidades que se consideran importantes para que el programa pueda ser utilizado por usuarios comunes del sistema, sin olvidar los objetivos de un sistema de migración de procesos.

1. `Guardar el estado de procesos.` El sistema deberá proveer la capacidad de, dado un proceso, crear una imagen mínima del estado del proceso para poder enviar. A diferencia de checkpointing, no se debe guardar toda la información del mismo, si no solo la necesaria para que el proceso se pueda seguir ejecutando si el contexto del sistema operativo es el mismo. Más específicamente, no se deben guardar los archivos abiertos (en cualquiera de sus formas) ni cualquier otra forma de estado externo al programa. Guardar el estado a disco es opcional en esta etapa.

2. `Guardar el estado de un proceso a disco.` Se deberá tener la opción de, dado un proceso, guardar su estado en disco para su posterior lectura. Se pretende de esta manera poder posteriormente agregar formas de checkpointing incremental y tolerancia a fallas.

3. `Leer y cargar el estado de un procesos desde disco.` El programa debe brindar la posibilidad de leer los archivos que ha guardado a disco para poder cargar un proceso.

4. `Matar y dejar de ejecutar otros procesos.` El sistema debe interactuar con el sistema operativo para dejar de ejecutar procesos, matarlos

y borrar cualquier residuo del mismo si es necesario. La razón de este requisito es poder dejar el sistema operativo sin problemas luego de migrar un proceso.

5. `Descubrimiento de otros sistemas.` Se debe implementar un algoritmo de descubrimiento de otros nodos siguiendo el esquema `peer-to-peer` para poder descubrir y comunicarse con otros sistemas. Se espera que esta sea la forma principal de decidir cuál nodo será el receptor de un proceso para el balanceo de carga. El descubrimiento de otros sistemas deberá servir también al sistema para armar una lista de compatibilidad utilizada en el momento de elegir el sistema receptor.

6. `Comunicación y negociación con otros sistemas.` El sistema tiene que comunicarse con otros sistemas para transmitir cualquier error o cualquier información que sea necesaria para mantener la red `peer-to-peer` en funcionamiento. Además, a través de esta comunicación deberá poder negociar con otros sistemas para decidir el balanceo de carga.

7. `Enviar estado del proceso.` Debe ser capaz de enviar el estado de un proceso hacia otro sistema sin tener que guardar el estado a disco. Previa comunicación con los otros sistemas, el sistema deberá enviar la mínima unidad indispensable para poder ejecutar el proceso seleccionado en el otro sistema. La información adicional será enviada a medida que el nuevo proceso la necesite.

8. `Recibir pedidos de envío de información adicional.` El sistema debe seguir un método Copy-On-Reference para el envío de información adicional del proceso, por lo que antes de enviar cualquier (y toda) otra cosa necesaria para el proceso, se debe esperar un pedido desde la máquina destino. Como funciona a alto nivel, el Copy-On-Reference solo hace referencia a archivos (en el sentido de UNIX) y no a páginas de memoria.

9. `Recibir pedidos para iniciar un nuevo proceso.` Para poder realizar la migración, es necesario que el sistema provea de un mecanismo para recibir pedidos para el inicio del proceso migrado en un sistema remoto.

10. `Iniciar y ejecutar nuevos procesos (procesos generados).` Se debe tener la posibilidad de iniciar un nuevo proceso con estado nulo, ya que se necesita tener un recipiente para la información necesaria para ejecutar un proceso migrado. Este proceso debe ejecutar en un ambiente controlado, actuando el sistema como un intermediario al sistema operativo.

11. `Agregar información a uno de los procesos generados.` Luego de iniciar un proceso, se debe poder insertar información en el proceso y proveer de toda la lógica necesaria para que el proceso generado pueda ejecutarse remotamente. Si es necesario para el correcto funcionamiento del programa, debe mantener abiertas conexiones con el sistema origen.

12. `Guardar información de procesos migrados que no han sido matados.` El sistema debe guardar toda la información de los procesos que no han sido matados para ser ejecutados remotamente. De esta manera interactuará como un intermediario en el sistema origen.

13. `Proveer de proxy, terminal gráfica remota e IPC si es necesario.` Si el proceso a ser migrado tiene abiertas conexiones a la red, el sistema debe actuar de proxy para las mismas. Si el proceso tiene interacción con la interfaz gráfica, se debe proveer al sistema origen de una interfaz gráfica para comunicarse con el proceso remoto. Si el proceso se comunica con otros, se deben mantener estas comunicaciones en el sistema origen y proveérselas al sistema remoto para el proceso generado.

14. `Virtualizar llamadas al sistema.` El sistema que inicia un nuevo proceso debe poder virtualizar ciertas llamadas al sistema para poder redirigir las llamadas que sean necesarias al sistema origen. De esta manera, se logra solo transmitir la información estrictamente necesaria.

15. `Tolerancia a fallas.` Un proceso deberá poder ser migrado completamente con todos sus archivos de manera eager-all si es necesario, ya sea por fallas en el sistema origen o remoto, o por razones administrativas.

16. `Interfaz administrativa.` Se debe proveer de una interfaz que permita realizar las siguientes acciones:

    a) Ver una lista de procesos migrados.

    b) Seleccionar un proceso para migrar.

    c) Seleccionar un proceso para guardar su estado.

    d) Seleccionar un archivo para cargar un proceso.

    e) Seleccionar un proceso para retornar desde el sistema destino posterior a una migración.

    f) Ver estadísticas en cuanto al uso del sistema por parte de sí mismo y procesos remotos.

17. `Procesos no migrables.` Los procesos del sistema operativo y los mismos procesos necesarios para el sistema serán considerados no migrables para esta implementación.

Con estos requisitos, se nota que el programa diseñado contaría con todas las propiedades y características principales de un sistema de migración de procesos orientado al usuario. Se apunta con estos llegar a una facilidad de uso y a una mínima interacción del sistema, a través de un programa cliente bien definido que permite el mayor control por parte del usuario, al mismo tiempo que mantiene su simplicidad. Es así como estos requisitos plantean las funcionalidades principales y mínimas (elegidas a partir de lo visto en otros sistemas y los objetivos de este trabajo) para que el diseño pueda partir de una base fundamental.

4.2. Requisitos no funcionales

Así como los requisitos funcionales plantean las características internas del programa y las funcionalidades que debe tener el sistema para cumplir con sus propósitos, es necesario también plantear los requisitos no funcionales para poder generar un contexto de ejecución en el cual el programa pueda ser ejecutado sin problemas. Es así que se deciden las características necesarias para que el programa cumpla con sus objetivos, además de características que permiten controlar los errores y entender más fácilmente el programa.

1. `Nivel de Usuario.` El sistema debe ser un programa en el nivel de usuario y realizar las llamadas necesarias al sistema para poder migrar procesos, conocer su estado y obtener cualquier otro recurso que se necesite según los requisitos funcionales. De esta manera, se logra que no se necesite modificar el kernel o el sistema operativo, incrementando la facilidad de uso del programa.

2. `Transparente.` Se debe implementar una migración transparente para el proceso y el usuario, de tal manera que ningún proceso sepa que ha sido migrado y que el usuario solo pueda obtener información de cuáles procesos han sido migrados y no a qué computadoras. La transparencia se considera necesaria puesto que para que el usuario pueda migrar cualquier proceso sin problemas, el sistema debe no cambiar el contexto del programa migrado de manera notable.

3. `Distribuido.` El sistema debe ser completamente distribuido y no mantener información en ningún servidor centralizado. Para lograr la primera comunicación de nodos, se puede tener una lista de direcciones conocidas que ejecutan el sistema. La distribución es una consideración necesaria para cualquier sistema de migración de procesos, como se ha notado en las secciones anteriores.

4. `Escrito en C.` El programa será escrito en su totalidad en el lenguaje C, tanto para mantener portabilidad como para lograr una mejor eficiencia. No se hacen consideraciones específicas en cuanto a los estándares de codificación.

5. `Formato del archivo de estado del proceso.` Se especifica al final de esta sección el formato que deben seguir los archivos que contengan la información de estado de un proceso.

6. `Errores.` Los errores del sistema deben ser documentados en su totalidad y deben ser avisados al usuario a través de un sistema de logs. De esta manera, se tiene un programa bien definido, que puede proveer de la seguridad que requiere un sistema que brinde tolerancia a fallas.

7. `Ejecutar con privilegios del usuario.` El sistema debe ejecutar con los privilegios del usuario que lo ejecuta y por lo tanto debe poder acceder solo a los procesos a los que este usuario tiene permisos. Se debe mostrar una advertencia al usuario en caso de ejecutar con el usuario de mayor privilegio. Este requerimiento es importante en cuanto a la seguridad del sistema, ya que evita problemas con la migración de procesos de otros usuarios.

Es importante notar que que la facilidad de usar el programa viene dada por el requisito funcional 16 y los requisitos no funcionales 1, 2, 6 y 7. Esto se debe a que para la realización de este programa se entiende facilidad como la cantidad mínima de cambios necesarios que se deben realizar en el ambiente para que el programa funcione correctamente.

Con todos estos requisitos planteados, incluyendo a los funcionales, se entiende el alcance del diseño y se puede empezar a pensar en el funcionamiento más interno del programa. Puesto que el sistema diseñado cumple con estos requisitos, se tienen todas las condiciones necesarias que debe cumplir el programa

para poder ser considerado un sistema fácil de usar para el usuario y que provea tolerancia a fallas a través de la migración de procesos.

4.3. Módulos

Como se entiende que el programa tiene un contexto más imperativo y funcional, se decide por una mirada modular del sistema. En esta sección se plantean los elementos que se consideran necesarios para el funcionamiento del programa, aunque estos pueden no ser vistos físicamente en el programa en ejecución. La idea principal es separar responsabilidad y poder dividir los problemas en situaciones más fáciles de comprender y solucionar.

Los módulos aquí descritos contendrán funciones que permitirán la relación entre ellos e implementarán en conjunto las distintas soluciones a los requisitos planteados. La elección de los mismos fue dada por tratar de encontrar las unidades mínimas indivisibles que se pueden ver entre los requisitos, dividiéndolos en clases de responsabilidades.

Es así que se plantean ocho distintos módulos o clases de responsabilidades: guardar un proceso, cargar un proceso, balancear el sistema, enviar un proceso, recibir un proceso, intermediar entre el proceso y el sistema operativo y viceversa, y la interface del sistema. A continuación se detallan más exactamente cada una de sus funcionalidades y responsabilidades.

4.3.1. saver

Este módulo se encarga de realizar todas las rutinas necesarias para guardar un proceso tanto sea a disco como a memoria. De esta manera, también es el encargado de matar procesos si es necesario. En sí, el módulo tiene como principales funciones: obtener información de procesos, guardar estado de procesos, obtener información adicional de los mismos si es necesario, y manejar el estado de los procesos. Este módulo debe cumplir los requisitos 1, 2 y 4.

Envía mensajes a: proxy(4.3.6)

Responde pedidos de: sender(4.3.4), interface(4.3.8), receiver(4.3.5)

4.3.2. loader

Debe cumplir con todo los objetivos especificados en los requisitos 3, 10 y 11, efectivamente convirtiéndose en el encargado de crear procesos que han sido migrados en el sistema destino. De esta manera, este proceso tiene que: iniciar procesos nulos, obtener la información y modificar el estado de los procesos generados y empezar su ejecución como procesos migrados.

Envía mensajes a: virtualizer(4.3.7)

Responde pedidos de: receiver(4.3.5), sender(4.3.4), interface(4.3.8)

4.3.3. balancer

Es el encargado de comunicarse con otros nodos de la red, aprender sobre sus capacidades y sus estados, al mismo tiempo que obtiene información sobre el propio nodo donde se ejecuta y lo envía. De esta manera, se encarga de los requisitos 5, 6, 15, 17 cumpliendo las siguientes funciones: mantener una lista de nodos, iniciar conexiones, recibir conexiones, terminar conexiones, comunicar necesidades e información de balanceo de carga, obtener cuáles procesos es mejor

migrar y asegurar que si el equipo se necesita apagar o tiene problemas, se reenvíen todos los procesos a sus sistemas origen.

Envía mensajes a: balancer remoto(4.3.3), receiver(4.3.5), sender(4.3.4)

Responde pedidos de: sender(4.3.4), balancer remoto(4.3.3), interface(4.3.8)

4.3.4. sender

Responsable de realizar el envío de procesos a otros sistemas y de mantener información sobre los procesos migrados, cumpliendo los requisitos 6, 12 y 7. Es así que se encarga de que se realicen las siguientes funcionalidades: enviar procesos a migrar hacia otros sistemas, recibir pedidos de los otros sistemas por información de un proceso, guardar toda la información que no haya sido enviada para poder enviarla posteriormente, y obtener toda la información de los procesos que vuelvan al sistema origen.

Envía mensajes a: receiver remoto(4.3.5), balancer(4.3.3), proxy(4.3.6), saver(4.3.1), loader(4.3.2)

Responde pedidos de: receiver remoto(4.3.5), balancer(4.3.8)

4.3.5. receiver

Como encargado de recibir pedidos de procesamiento, implementa los requisitos 6 y 9. Sus funciones principales son: recibir información básica del proceso, pedir y recibir información adicional, mantener información sobre los procesos remotos y enviar todos los procesos en caso de falla.

Envía mensajes a: sender remoto(4.3.4), loader(4.3.2), saver(4.3.1)

Responde pedidos de: balancer(4.3.3), sender remoto(4.3.4), virtualizer(4.3.7)

4.3.6. proxy

Maneja todos los recursos extra necesarios que no han sido migrados, ya sea comunicación con proceso en el sistema origen, mantener conexiones de red que están siendo usadas y no pueden ser cerradas o terminadas y mantener una terminal gráfica si el proceso migrado lo requiere. El requisito que implementa es el 13.

Responde pedidos de: sender(4.3.4), saver(4.3.1)

4.3.7. virtualizer

Se encarga de iniciar nuevos procesos y de virtualizar todas las llamadas al sistema que sea necesarias redirigir al sistema origen. Por lo tanto, cumple con los requisitos 10 y 14. Es uno de los módulos críticos para poder hacer funcionar un proceso en otro sistema.

Envía mensajes a: receiver(4.3.5)

Responde pedidos de: loader(4.3.2)

4.3.8. interface

Provee de una interfaz, tanto sea por línea de comando como por gráficos, para cumplir con lo descrito en el requisito 16.

Envía mensajes a: balancer(4.3.3), saver(4.3.1), loader(4.3.2)

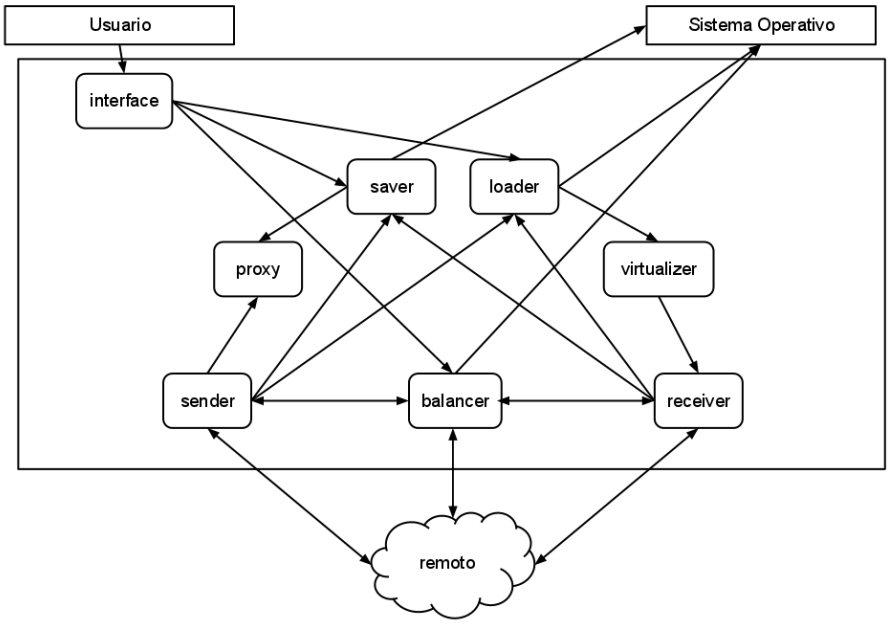

Figura 1: Relaciones entre los módulos

4.4. Arquitectura

La Figura 2 presenta una mirada básica de los sistemas que participan en una migración, sus distintas comunicaciones y negociaciones. Como es un sistema distribuido peer-to-peer, solo se presenta la mirada desde el sistema origen y suponiendo que solo ocurre una migración de procesos. Se espera que de esta manera se puedan explicar un poco mejor las interconexiones y la migración.

4.5. Flujos del programa

Al tener definida la separación de responsabilidades y la comunicación entre los mismos, se pretende a continuación indicar los distintos casos de uso que se le puede dar al programa y que el sistema debe poder manejar. Es por eso que se decide describir distintos flujos del programa en cuanto al uso y a su interacción con el usuario, tanto para tener una base para la implementación como para entender en forma rápida como debería funcionar internamente el programa.

Los flujos que se presentan no pretenden en sí ser exhaustivos ni demasiado estrictos en cuanto a sus pasos y a sus funcionalidades, pero sí se quiere que se entiendan como la base sobre la cual las distintas acciones que se realizan con el programa se pueden llevar a cabo. Con los flujos que se explican a continuación se brinda un paso a paso más detallado de un sistema distribuido de migración de procesos que cumple con los requisitos antes planteados.

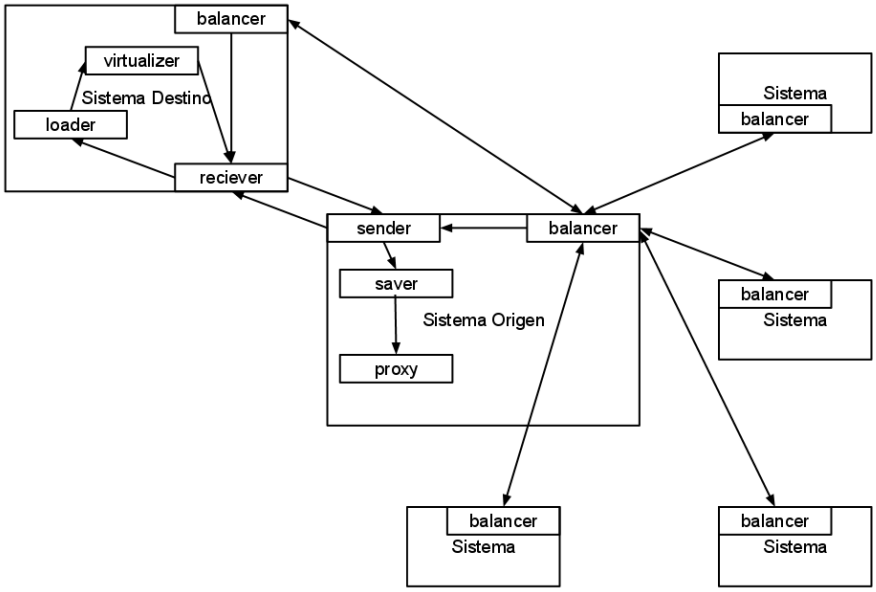

Figura 2: Relaciones entre sistemas

4.5.1. Obtener información del sistema

Obtener la información de los procesos del sistema para así calcular si hay que realizar una migración de procesos. Guardar información pasada para estadísticas.

- `Encargados:` balancer
- `Flujo:`
  1. Recibir del sistema operativo información de los procesos
  2. Guardar información de los procesos en memoria
  3. Calcular el estado del sistema
  4. Si se decide migrar un proceso, pasar al flujo Migrar proceso
  5. Actualizar estadísticas del sistema

4.5.2. Descubrimiento de nodos

Descubrir otros nodos en la red que utilicen el sistema para guardarlos y empezar comunicación con ellos

- `Encargados:` balancer
- `Flujo 1: Enviar pedidos`
  1. Avisar a todos los nodos conocidos su propia lista de nodos conocidos

2. Recibir respuestas de nodos que conocen
3. Actualizar lista de nodos conocidos

- `Flujo 2: Recibir pedidos`

1. Recibir pedidos de nodos conocidos
2. Enviar lista de todos los nodos conocidos localmente exceptuando aquellos recibidos
3. Actualizar lista de nodos conocidos con aquellos que no estaban

4.5.3. Balanceo de carga

Obtener información de otros nodos y actualizar las estadísticas para el balanceo de carga

- `Encargados:` balancer
- `Flujo 1: Enviar pedidos`

1. Pedir a la lista de nodos conocidos información sobre su estado
2. Actualizar estructuras en memoria con la nueva información y las propias estadísticas

- `Flujo 2: Recibir pedidos`

1. Recibir pedido de información
2. Enviar información del nodo

4.5.4. Migrar proceso

Lograr la migración en sí de un proceso a partir de un sistema, incluyendo el guardado, la transferencia y el cargado del mismo.

- `Encargados:` balancer, sender, saver, proxy, receiver, loader, virtualizer
- `Flujo:`

1. balancer decide a través de estadísticas qué proceso migrar
2. balancer calcular nodo destino
3. balancer realiza pedido a nodo destino
4. balancer remoto decide si aceptar el pedido y avisa a balancer local. Si no, se envía nueva información de carga
5. balancer local recibe autorización para realizar la migración. Si no, se vuelve al paso 2.
6. balancer envía identificador del proceso y dirección del nodo destino a sender
7. sender envía a saver identificador del proceso
8. saver realiza una interrupción al proceso y obtiene toda la información

9. saver guarda toda la información del proceso y envía la cantidad mínima al sender junto con un nuevo identificador local del proceso. Si es necesario, inicia proxy
10. proxy recibe de saver toda la información del proceso y mantiene abiertos los recursos necesarios
11. sender guarda identificador local del proceso junto al nombre del mismo y otro información pertinente y envía la información mínima al receiver remoto, indicando si se envió toda la información necesaria
12. receiver remoto recibe pedido y reenvía pedido a loader
13. loader inicia nuevo proceso y carga información. Si no se tiene toda la información, inicia virtualizer para el proceso. Se devuelve a receiver el identificador remoto nuevo del proceso y/o el virtualizer.
14. virtualizer inicia todas las funciones necesarias e inicia el proceso
15. receiver obtiene la información, la guarda localmente y envía el identificador al sender indicándole que el proceso se inició sin problemas
16. sender recibe información del proceso y guarda localmente el identificador remoto y el nodo destino

- `Notas:` Si en algún momento se pierde conexión al nodo destino, se empieza nuevamente. Si el nodo destino no logra comunicarse nuevamente con el nodo origen, guarda el proceso a disco como se explica en 15. Este flujo puede ser originado por interface.

4.5.5. Ejecución remota

Ejecutar un proceso remotamente sin tener migrados todos los recursos o con recursos que no puedan ser migrados

- `Encargados:` sender, proxy, receiver, virtualizer
- `Flujo:`
  1. virtualizer recibe del proceso generado pedido de recurso que no se encuentra en la información disponible
  2. virtualizer envía pedido con su identificador a receiver para enviar pedido al recurso
  3. receiver correlaciona el identificador con el proceso y el nodo origen, y envía pedido al recurso a través del sender
  4. sender recibe pedido y envía al proxy el identificador y pedido al recurso
  5. proxy realiza el pedido al recurso y envía al sender toda la que considera útil en la transmisión
  6. sender envía información a receiver remoto
  7. receiver recibe la nueva información y la redirige al virtualizer
  8. virtualizar envía la información al proceso

4.5.6. Tolerancia a fallas

Si falla el proceso en el sistema remoto o el sistema remoto no va a poder seguir ejecutando el proceso, reenviar el proceso al sistema origen.

- `Encargados:` balancer, receiver, saver, loader, sender, virtualizer
- `Flujo 1:`
    1. balancer decide que lo mejor es migrar un proceso de otro sistema, o virtualizer nota errores en el proceso generado, o receiver recibe pedido de devolver proceso
    2. balancer envía pedido a receiver para matar a virtualizer, o virtualizer envía pedido a receiver para migrar proceso
    3. receiver mata al virtualizer y pide a saver el identificador del proceso
    4. saver guarda toda la información del proceso local y lo mata, devolviendo la información a receiver
    5. receiver envía toda la información al sender remoto relacionado con el proceso
    6. Si el envío falla, receiver envía señal a saver para que guarde toda la información a disco
    7. sender recibe información de proceso y se la redirige a loader con la información antigua del proceso
    8. loader intenta recrear el proceso de la manera más parecida a la que se encontraba anteriormente
    9. sender mata al proxy si este se había creado
- `Flujo 2:`
    1. balancer nota que nodo deja de comunicar y avisa a sender, o interface recibe pedido de retornar un proceso
    2. sender busca si hay procesos migrados y envía pedidos al receiver correspondiente para que devuelva los procesos
    3. si receiver no contesta, sender llama a loader con la última información que se tiene del proceso
    4. sender obtiene toda la información del proceso y la reenvía al loader
    5. loader recrea el proceso de la manera más parecida posible
    6. sender mata al proxy si este se había creado
- `Notas:` El flujo 1 se puede realizar repetidas veces sin matar los procesos, para mantener al sender con toda la información necesaria para recrear el proceso.

4.5.7. Guardar proceso a disco

Guardar un proceso a un archivo de disco

- `Encargados:` interface, saver

- `Flujo:`
    1. interface recibe pedido para guardar proceso y lo reenvía a saver
    2. saver guarda toda la información del proceso en el archivo tratando de no frenar su ejecución

4.5.8. Cargar proceso a disco

Cargar un proceso desde un archivo de disco

- `Encargados:` interface, loader
- `Flujo:`
    1. interface recibe pedido de cargar archivo y lo reenvía a loader
    2. loader carga toda la información en un nuevo proceso local

4.5.9. Lista de proceso y estadísticas

Obtener información sobre los procesos y estadísticas del sistema

- `Encargados:` interface, balancer
- `Flujo:`
    1. interface recibe pedido y lo reenvía a balancer
    2. balancer devuelve la información y la envía a interface
    3. interface muestra información en pantalla

4.5.10. Inicio del sistema

Reiniciar procesos que hayan sido guardados a disco por tolerancia a fallas

- `Encargados:` loader
- `Flujo:`
    1. Si loader encuentra archivos guardados a disco por tolerancia a fallas, se inicializan y si es necesario se crea virtualizer
    2. loader envía información a receiver
    3. receiver trata de comunicarse con sender remoto. Si no es posible, se mata el virtualizer
    4. sender responde afirmativamente y carga información necesaria
    5. receiver obtiene información y avisa a loader

4.5.11. Cierre del sistema

Guardar toda la información de los procesos si el sistema se cierra o pedir que todos los procesos se devuelvan al sistema origen.

- `Encargados:` receiver, loader, sender, saver
- `Flujo:`
  1. receiver envía orden a saver para guardar todos los procesos remotos a disco y empieza en el flujo Tolerancia a fallas
  2. sender envía orden a todos los nodos destino de devolver proceso y empieza en el flujo Tolerancia a fallas si no recibe respuesta.

4.6. Comunicación

La comunicación del sistema se realizará para simplificar el problema sobre la capa TCP/IP, y no se especifica un protocolo de comunicación específico debido a la poca información que debe ser enviada. Se nota sin embargo, que toda la información que se enviaría entre los módulos balancer sería en modo texto plano, mientras que la imagen del proceso se enviaría en binario, de la misma manera que es guardada en disco.

4.7. Especificaciones

Para guardar la información se utilizarán estructuras internas del programa que contendrán toda la información necesaria de un proceso para poder ser reiniciado. Se guarda entonces de manera recursiva la estructura principal del proceso guardado y las demás estructuras que esta referencie. En sí, la imagen del proceso debe contener por lo menos el estado de los registros del sistema cuando el proceso fue frenado, la disposición de la memoria y toda la información contenida en ella, y cualquier otro descriptor necesario para que el proceso pueda ser reanudado (como por ejemplo, archivos abiertos).

La estructura principal del proceso contendrá entonces toda la información general del proceso y luego una lista de pedazos de información de distintos tipos que será escrita en memoria de manera secuencial. Es decir, la estructura principal contendrá distintas partes del proceso guardadas en una lista, donde cada parte debe describir un tipo y tener toda la información necesaria para describir la parte del programa. De esta manera se tendrán partes que son pedazos de memoria, con su disposición y el contenido de la misma, y otras partes que tendrán la información de los registros. Se deja a la implementación los detalles de esta estructura.

4.8. Pruebas

Se realizarán pruebas básicas para comprobar el funcionamiento correcto del programa. Los programas que se utilizarán tendrán una mezcla entre llamadas al sistema y una implementación propia de iteración y recursión. El ejemplo más básico es escribir en pantalla los números del 1 al 100, esperando un segundo antes de escribir el siguiente número. Estas pruebas deben funcionar, pero no se tendrá en cuenta si el rendimiento de la migración es mejor que el rendimiento local.

5. Implementación del prototipo

Para validar el funcionamiento del diseño se decidió implementar un prototipo vertical que cumpla con algunos de los requerimientos, especialmente aquellos más relacionados al área de migración de procesos; de esta manera se eligieron los requerimientos que se cree tienen más posibilidad de demostrar tanto la utilidad como la posibilidad de crear buenos sistemas de migración de procesos a nivel de usuario. En el Cuadro 2 se presentan los requisitos implementados y aquellos que se consideran importantes para una futura implementación de un prototipo más avanzado.

Los módulos implementados en el prototipo para cumplir con estos requisitos son archivos que contienen funciones a través de archivos de cabecera de C. Cada módulo tiene el mismo nombre que en la parte de diseño e implementan funciones que pueden ser reutilizadas si es necesario por otros módulos y además permiten una fácil extensión posterior del programa. Los módulos implementados son: balancer(4.3.3), loader(4.3.2), saver(4.3.1), sender(4.3.4) y virtualizer(4.3.7).

5.1. Utilidades del sistema operativo

Al realizar un sistema de migración de procesos sobre GNU/Linux, es importante notar aquellas utilidades importantes que se manejaron en este prototipo y describir su uso. Se explican las tres partes más notables que el sistema utiliza: el `procfs`, la utilidad `ptrace` y el uso de la red; además por último se explica el funcionamiento de la arquitectura de computadoras y como puede ser usado por el prototipo. Así quedan detallados el funcionamiento de estas propiedades del sistema y el uso que el prototipo les da a las mismas.

5.1.1. Sistema de archivos `/proc` (`procfs`)

El sistema de archivos `procfs` es un sistema de archivos especial de algunos sistemas operativos basados en UNIX (incluyendo GNU/Linux) que permite obtener información tanto de los procesos que están actualmente ejecutando en una computadora como de cierta información sobre la ejecución del sistema. Usualmente se encuentra en la ruta /proc y tiene en esta ruta archivos generales del sistema que pueden ser leídos rápidamente y que además se mantienen siempre actualizados por el sistema operativo. Dentro de la carpeta, además hay una carpeta por cada proceso en ejecución del sistema que tiene de nombre el identificador del proceso y que contiene archivos que ayudan a entender el estado del mismo. Se pasan a explicar los archivos más notables para la migración de procesos, especialmente para la explicada en este trabajo.

En la carpeta fd se tiene por cada archivo abierto por el programa un enlace simbólico con el nombre del descriptor de archivos, apuntando al archivo en sí que se tiene abierto, lo cual facilita la lectura del archivo para poder guardar su estado. Esta carpeta, junto a la carpeta fdinfo que contiene archivos con información sobre los archivos que se tienen abiertos (como por ejemplo la posición del puntero), permite a un proceso externo poder guardar toda la información de los descriptores de archivos que tiene un proceso y de esta manera poder restaurarlos.

Especialmente importante para este prototipo, dentro de la carpeta de un proceso, existe un archivo llamado `maps` que contiene toda la información sobre

Requisito	Prototipo	Prototipo avanzado	Sistema final
Requisitos funcionales			
1. Guardar el estado de procesos	Sí	Sí	Sí
2. Guardar el estado de un proceso a disco	Sí	Sí	Sí
3. Leer y cargar el estado de un procesos desde disco.	Sí	Sí	Sí
4. Matar y dejar de ejecutar otros procesos	Sí	Sí	Sí
5. Descubrimiento de otros sistemas	No	No	Sí
6. Comunicación y negociación con otros sistemas	No	No	Sí
7. Enviar estado del proceso	Sí	Sí	Sí
8. Recibir pedidos de envío de información adicional	No	Sí	Sí
9. Recibir pedidos para iniciar un nuevo proceso	Sí	Sí	Sí
10. Iniciar y ejecutar nuevos procesos (procesos generados)	Sí	Sí	Sí
11. Agregar información a uno de los procesos generados	No	Sí	Sí
12. Guardar información de procesos migrados que no han sido matados	Sí	Sí	Sí
13. Proveer de proxy, terminal gráfica remota e IPC si es necesario	No	Sí	Sí
14. Virtualizar llamadas al sistema	No	Sí	Sí
15. Tolerancia a fallas	No	Sí	Sí
16. Interfaz administrativa	Sí	Sí	Sí
17. Procesos no migrables	No	No	Sí
Requisitos no funcionales			
1. Nivel de Usuario	Sí	Sí	Sí
2. Transparente	Sí	Sí	Sí
3. Distribuido	Sí	Sí	Sí
4. Escrito en C	Sí	Sí	Sí
5. Formato del archivo de estado del proceso	Sí	Sí	Sí
6. Errores	Sí	Sí	Sí
7. Ejecutar con privilegios del usuario	Sí	Sí	Sí

Cuadro 2: Requisitos del prototipo y del sistema final

el VMA del mismo, lo que permite poder obtener las direcciones que el proceso usa y otra información de cada área, que permite al proceso que guarda la información poder obtener toda la memoria que el proceso está utilizando. Para esto se debe leer cada línea de este archivo y entender la información que se encuentra en el mismo; cada línea corresponde a un área de memoria distinta, y como se explica en la sección 1.6, contiene también toda la información sobre las bibliotecas dinámicas, de manera que pueden ser restauradas en el proceso generado.

Por último, se nota que hay consideraciones que aunque se trataron de tomar en cuenta, por razones de simplicidad fueron obviadas. Una de ellas es que las bibliotecas pueden no ser enviadas si en la computadora destino se tiene la misma versión (ya que se podrían levantar directamente desde disco); otra es que no se hace un chequeo de integridad de los datos, ya que se supone que gracias a la confiabilidad de TCP/IP, los datos que se recibieron están en perfectas condiciones, aunque esto puede no ser del todo cierto.

5.1.2. `ptrace`

`ptrace` es una llamada del sistema de GNU/Linux que permite a un proceso poder investigar el estado de otro proceso, además de poder también modificarlo. Esta llamada en realidad funciona directamente sobre los procesos hijos de un proceso, pero provee una llamada específica para convertir cualquier otro proceso en un proceso hijo del proceso actual. Se presentan algunas de las funcionalidades más importantes de esta utilidad.

Una de las opciones más importantes es poder capturar la memoria de otro proceso, y para eso se brinda la posibilidad de tomar un proceso y convertirlo en proceso hijo. Esto se realiza a través del identificador del proceso y permite que las demás utilidades de esta llamada del sistema puedan funcionar correctamente. De todas maneras, el proceso que se convierte en hijo no tiene información sobre el proceso padre y sigue suponiendo que el proceso padre de este es el original. De manera opuesta, se tiene también una opción para que un proceso permita a otro proceso utilizar `ptrace`.

Con esta llamada también se pueden mandar ciertas señales a otro proceso, permitiendo tanto frenar el otro proceso como continuarlo desde donde quedó. Existe también la posibilidad de matar al proceso hijo, de tal manera que su ejecución termine. Esta utilidad es necesaria para poder obtener el estado del proceso en un determinado momento y poder matarlo cuando el proceso ya ha sido migrado.

También permite el acceso a la memoria del proceso hijo pasándole a la llamada la dirección de memoria que se quiere utilizar. De esta manera se permite que el proceso que utiliza `ptrace` pueda tanto leer las palabras que el proceso tiene guardadas en memoria como modificarlas. Es así que el proceso que guarda el estado del proceso puede obtener todos los datos de memoria, pero esto también permite que otro proceso pueda inyectar datos y hasta código en otro programa, pudiendo ser considerado un problema de seguridad, como se explica en [41].

La última opción importante para la migración de procesos es la posibilidad de obtener en una estructura el estado de los registros del proceso hijo, de esta manera pudiendo obtener una de las partes más importantes del estado de un proceso. Asimismo, se posibilita modificar los registros que pueden ser de

mucha utilidad para restaurar el proceso o, como utiliza CryoPID, para realizar llamadas del sistema y obtener sus respuestas como si el proceso padre fuera el proceso hijo.

De esta manera, esta llamada al sistema es muy importante para obtener información sobre otros procesos, tanto así que los debuggers más comunes la utilizan tanto para conocer el estado de la memoria y de los registros como para poder realizar breakpoints dentro del código del otro proceso. También para un sistema de migración de procesos es necesario hacer uso de esta llamada para conocer completamente el estado de otro proceso.

5.1.3. Redes

Aunque de manera menor en este prototipo, es importante entender el uso de los sockets de GNU/Linux para poder comunicarse con otros procesos y con otras computadoras, de tal manera de enviar la información sobre el estado de un proceso y de esta manera migrarlo. Además, aunque la funcionalidad no fue implementada, se necesita saber la capacidad de las redes para poder realizar la estructura P2P mencionada anteriormente.

En el prototipo se utiliza el protocolo TCP sobre IP para lograr una comunicación confiable y con control de errores. El protocolo de aplicación creado no merece mención en esta sección ya que solo envía los datos del proceso sin preocuparse por datos de control ni pedidos de migraciones, por lo que no tiene ningún protocolo especificado. Al utilizar IP, las direcciones que recibe el programa son de este protocolo.

5.1.4. Arquitectura de computadores

La arquitectura de los computadores recibe especial atención en todos los programas que deben ser muy optimizados o que deben tratar de proveer funcionalidades que tienen más sentido en el espacio del kernel. De esta manera, una migración de procesos de usuario, aunque tratando de ser portable, necesita conocer la arquitectura y proveer la mayor funcionalidad del sistema en este sentido.

Es así que dentro del código de este prototipo se utilizan las arquitecturas x86 y x64 para generar un código de máquina que pueda ser entendido por la computadora. Como se explicará en la funcionalidad del sistema, este código de máquina es necesario debido a limitaciones del kernel y a funcionalidades que debe tener el sistema, que de otra manera no podrían ser creadas.

Sin embargo, es importante entender que la mayoría del código debe ser en C para facilitar la utilización del sistema en distintas computadoras y para poder ser extensible. Aunque la arquitectura juega un papel importante incluso en los cabezales de los ejecutables en formato `ELF` (Executable and Linkable Format) [42], se intentó evitar esta parte, entre otras cosas, para minimizar la utilización de código dependiente de la arquitectura.

5.2. Decisiones de implementación

Para poder implementar correctamente el programa, se tomaron decisiones para facilitar el desarrollo y para tomar ideas que funcionan en otros programas. De esta manera, la primer decisión fue tomar como referencia de implementación

el programa `CryoPID`(explicado en 3.1.4), ya que tiene como objetivos ser un proceso de nivel de usuario que permita realizar checkpoints de procesos en gran variedad de casos y reanudarlos tanto en la computadora actual como en otra. Sin embargo, como este programa esta desactualizado y no funciona de manera correcta en versiones actuales del kernel Linux, se debieron realizar varias modificaciones para el correcto funcionamiento y para aprovechar nuevas funcionalidades del kernel.

El programa CryoPID hace gran uso de la modificación de cabezales del formato ejecutable `ELF`, formato usado por GNU/Linux por defecto para todos los ejecutables, para poder realizar la inyección del código del proceso generado. Estas modificaciones permiten la creación de archivos de checkpoint ejecutables para reanudar el programa, pero por el otro lado tienen mucha dificultad en ser portables y traen muchos problemas a la hora de ser implementadas por la poca resistencia del formato `ELF` a los errores y a los cambios. Se decide por tanto no utilizar esta parte del programa para evitar dificultades en la implementación.

Es importante notar que para poder leer el espacio de memoria de otro proceso que no es hijo del proceso actual, se hace uso de la utilidad `ptrace`, que permite tomar un proceso del usuario y seguirlo ejecutando como si fuera hijo del proceso que llamó a la utilidad. De esta manera se puede estudiar su memoria, registros, archivos abiertos y cualquier otra información que identifique el proceso. Sin embargo, en algunas distribuciones de GNU/Linux como Ubuntu [41], el uso de `ptrace` para convertir procesos externos en procesos hijos está bloqueado por razones de seguridad; considerando que estas distribuciones son pocas y que esta configuración puede ser cambiada, se opta por `ptrace`.

Para la reanudación de procesos se decidió utilizar código de máquina inyectado en una dirección de memoria que luego será ejecutado. De esta manera, el ejecutable que reanudará el proceso generado no utiliza configuraciones de compiladores y linkers complicadas que pueden dificultar el uso normal de la aplicación. Sin embargo, utilizar código de máquina implica poca portabilidad entre distintas arquitecturas, por lo que se intenta minimizar este código y que utilice, siempre que sea posible, llamadas a sistema para estandarizarlo.

Para la implementación del prototipo se decidió no realizar ninguna funcionalidad que permita el balanceo de los procesos. Aunque esto significaría una buena prueba del verdadero funcionamiento del sistema en cuanto a mejora del aprovechamiento de los recursos, también significa una capa de complejidad significante que, aunque puede ser fácilmente implementada, quita la concentración sobre la migración de proceso a nivel de usuario e implica pruebas de rendimiento que no deberían ser importantes en un prototipo.

Como el sistema operativo utiliza VMAs, si no se modifican las variables que apuntan a memoria, se deben restaurar todas las áreas de memoria en la dirección exacta del VMA en las que se encontraban; como es más fácil y simple restaurar las secciones de memoria al área correcta que encontrar todas las variables y modificarlas, se opta por esta opción.

Además, es de suma importancia notar que el proceso puede pedir memoria a través de la llamada sistema `mmap`, lo que significa que se genera una nueva área de memoria que el proceso puede referenciar con los permisos que se pidan. Este hecho junto con la dificultad que se tiene en diferenciar distintas áreas de memoria, se suma a la decisión tomada de que la memoria debe ser restaurada exactamente como está descrita en el VMA para que el proceso generado no encuentre dificultades en su ejecución.

Como las bibliotecas pueden tener distintas versiones y ser distintas en distintos sistemas, se ha optado en este prototipo que el área de memoria de una biblioteca sea copiada enteramente en la memoria del proceso generado. Esto significa que esta área de memoria no se va a compartir con otros procesos y no se va a tener por lo tanto una de las ventajas de una biblioteca dinámica. Sin embargo, esta decisión logra que sea más fácil restaurar el programa ya que con seguridad se tienen todas las funciones necesarias por el proceso en la dirección de memoria exacta a la que referencia.

Aunque se realiza una transmisión de archivos abiertos (ya se su contenido o solo su dirección en el sistema de archivos), a diferencia del diseño, no se mantiene un proxy para que se puedan seguir modificando los archivos localmente. De todas maneras, si se tiene un sistema de archivos compartidos, el proceso remoto podría seguir modificando o leyendo el mismo archivo que utilizaba en la computadora origen.

5.3. Representación de un proceso

Para el funcionamiento del programa, se crearon estructuras internas que el programa maneja para acomodar toda la información sobre un proceso. Estas estructuras en su mayoría fueron tomadas de CryoPID y se les modificó el nombre y algunos de sus atributos. Se decidió aunque todavía no se utilice, mantener todas las estructuras que describen el proceso para luego poder extender el programa más fácilmente.

La primera estructura a notar es `proc_process`, que es la estructura raíz; es decir, aquella que contiene toda la información de un proceso y tiene referencias a otras estructuras. En esta estructura se guarda el nombre de proceso (que en el prototipo actual no se utiliza) y una lista de partes del mismo. Esta lista es la que contiene todo el resto de los datos del proceso, por lo que siempre que se necesita, se itera sobre sus elementos.

Cada elemento de la lista es un `proc_chunk`, que es una estructura que puede tener dentro de ella cualquiera de las distintas estructuras. De esta manera se puede guardar el tipo de la parte del proceso (es decir, si es un registro o si es parte de la VMA) y la información en sí, lo que hace más fácil la iteración. Esta solución se prefirió sobre otras debido a que es la más fácil de extender.

Las otras dos estructuras que son importantes de mencionar son `proc_vma` y `proc_regs`. La primera es la que contiene la información sobre una parte de la memoria (que equivaldría a una línea del archivo `maps` en `procfs` y a un área de memoria del proceso) y contiene de esta manera todos los campos identificatorios e informativos sobre la misma: comienzo del área de memoria, largo del área, protección, banderas (si es compartida o no), si es una biblioteca dinámica o el heap, el nombre del archivo al cual está relacionado el área (si lo hay) o el nombre mismo que el sistema le da al área, y por último todos los datos que se encuentran en esa dirección de memoria.

La segunda estructura mencionada, `proc_regs`, contiene la información del estado de los registros de cuando se le hizo el checkpoint al proceso. De esta manera, el único atributo notable es un estructura user que es la que `ptrace` utiliza para devolver la información de los registros.

Por último, se utiliza la estructura `proc_file` y la estructura `proc_fd` para guardar la información de los archivos abiertos, el puntero a la posición, y el contenido del archivo.

Todas estas estructuras son luego escritas a memoria directamente, iterando sobre la lista, de manera que obtener la información de un proceso es simple y además manteniendo un tamaño relativamente pequeño para el archivo de checkpoint y para enviar el proceso.

5.4. Funcionamiento general

En esta sección se va a presentar el funcionamiento interno del programa para poder luego explicar las dificultades y las funcionalidades del mismo. No pretende ser una explicación extensa de las funciones y del código del programa, sino brindar una idea general de cómo se creó el programa y cómo se podría reproducir y mejorar. Se intentará solo explicar las partes relevantes a la migración de un proceso, como si fuera el flujo del programa. Por lo tanto, primero se explicará de forma rápida la interfaz que tiene el programa, para luego explicar cómo ejecutar algún ejemplo y de esta manera poder ver el funcionamiento del programa. Se entregan además las páginas man del programa para que sean leídas como un documento más técnico de lo que aquí se explica.

5.4.1. Interfaz

El prototipo creado cuenta con dos programas, procs y procsd; el primero sería el programa cliente y el segundo es un daemon que permite tener el programa ejecutando en todo momento para recibir procesos. Con el primer programa se pueden realizar todas las acciones de checkpoint localmente, como por ejemplo hacer un checkpoint en un archivo y restaurarlo posteriormente del mismo; de esta manera se tiene cierta migración de archivos manual, ya que el archivo en el cual se guarda el proceso puede ser enviado a otra máquina a través de cualquier otro método y restaurado en la máquina destino (siempre y cuando la otra computadora comparta la misma arquitectura).

El daemon abre dos sockets: con uno abre un puerto de Internet (que por defecto es el 12092) con el que se recibirán los procesos remotos y con el otro abre un socket UNIX con el cual se recibe toda la información que pida el proceso cliente. En este daemon (que tiene la opción de ser corrido sin daemonizar), se guardan en memoria todos los procesos recibidos remotamente a través de un id que pueden ser listados con el programa cliente, y además todos los procesos que se pidan guardar en el daemon desde el proceso cliente.

Los dos programas tienen varias opciones para realizar mejor el envío, como por ejemplo que el proceso que sea guardado no se mate y que siga corriendo localmente (que puede ser útil para guardar un proceso que demore mucho tiempo y se quieran tener checkpoints por si ocurre algún error inesperado). Además, el daemon utiliza syslog para informar de posibles problemas y de lo que actualmente se encuentra haciendo.

5.4.2. Ejecución

Para la ejecución de la migración de procesos, primero se debe ejecutar procsd, ya sea como daemon o no. Se pueden revisar los logs para ver si ocurrió algún problema. Luego de esto, en la computadora origen se utiliza el comando send del programa procs, con el programa que se debe enviar (ya sea desde un

archivo, directamente con el PID o el nombre del proceso, o desde un identificador provisto por el procsd local) y el host al que se debe enviar. De esta manera, si todo termina correctamente, en el daemon procsd de la computadora destino quedará guardado con un nuevo id el proceso que se envió.

Para restaurar el programa se tienen dos opciones: se puede correr el programa cliente con la opción receive sin ningún parámetro antes de enviar el proceso, lo cual hace que el programa espere a que llegue un nuevo daemon a la computadora destino para reanudarlo; o se puede enviar el proceso y luego reanudarlo con receive o también con la opción restore.

5.4.3. Interno

En esta sección se explicará de manera más detallada, la implementación del programa y el desarrollo que lleva a cabo cuando se guardan los datos de un proceso, se envían y se restauran. Como fue explicado anteriormente, trata de ser más bien una introducción a las utilidades que se aprovechan más que una explicación detallada del código del programa, por lo que algunas partes pueden estar demasiado simplificadas para ser más entendibles.

Cuando se recibe la opción de guardar un proceso en el programa cliente procs, lo primero que se realiza es obtener el PID del mismo para poder obtener toda su información, a no ser que ya se le haya pasado directamente como parámetro. Si no, se utiliza una combinación de las utilidades `ps` y `awk` para poder obtener el identificador del proceso a través de su nombre. Esta elección se tomó porque estos programas ya están muy bien optimizados para realizar este tipo de búsquedas y la implementación de nuestro propio sistema podría llegar a tener otros problemas.

Luego de obtenido de esta manera el identificador, se utiliza `PTRACE_ATTACH` para ejecutar el proceso indicado como proceso hijo del actual y poder leer toda su información. Se crean entonces vacías las estructuras y se inicializan sus valores, para luego esperar hasta que el proceso hijo esté frenado, ya que si no, no se puede utilizar `ptrace` para leer su memoria ni sus registros y hasta estos podrían cambiar y el estado del proceso quedaría corrupto.

Si la arquitectura del proceso que está ejecutando es de 32 bits, se guarda la información sobre el `TLS` (Thread Local Storage, donde se guarda información local a cada hilo), para poder luego restaurarlo. En 64 bits esto no es necesario ya que esta información está guardada directamente en los registros.

Posteriormente, entonces, lo que se hace para obtener el estado del proceso es leer su archivo `maps` en el `procfs`, que contiene todas las áreas VMA del mismo (el archivo se encuentra en la carpeta que tiene de nombre el identificador del proceso dentro de la carpeta /proc). Se itera sobre cada una de las líneas, parseando la información y leyendo sus datos para llenar la estructura explicada anteriormente (como la dirección de memoria de comienzo, el largo y el nombre del área). Con esta información se puede saber si el área es un heap o una biblioteca compartida (mirando el nombre, si existe). Se agrega entonces por cada línea una estructura explicando el área del proceso a la lista de la estructura raíz del proceso.

Después que se obtiene toda la memoria del proceso, se intentan encontrar los archivos abiertos. Para esto se lee el directorio fd del `procfs`, donde se tiene un archivo por cada descriptor. Se utiliza `stat` para obtener información sobre los archivos y luego se obtiene la dirección del mismo en el sistema de archivos.

Esta se utiliza para leer los contenidos del mismo si es necesario. Finalmente, se lee el directorio fdinfo, también de `procfs`, para obtener la posición del puntero al archivo.

Por último, se obtiene la información de los registros utilizando `PTRACE_PEEKUSER`, que permite obtener la memoria en donde están guardados los registros. Se intentó mejorar el programa utilizando la opción que tiene `ptrace` para obtener directamente los registros, pero ocurrían más errores por lo que se mantuvo la opción anterior. Se itera entonces sobre el tamaño de la estructura user que es la que va a contener los registros y se pide con `ptrace` todas las palabras en memoria de lo que sería esa estructura; de esta manera, se agrega a la lista del proceso la información de los registros contenida en una estructura user.

Es importante notar que todo esto solo se lleva a cabo cuando se quiere guardar el proceso. Si al comando send se le pasa la opción para que lea desde un archivo, se leen todos los datos guardados en ese archivo en una estructura nueva de proceso; y si lo que se pide es que se lea desde un id local, se pide al daemon local que pase al cliente la estructura del proceso referenciada por ese id.

Al tener la estructura que contiene toda la información del proceso, el programa entonces abre una conexión hacia el daemon remoto para enviar los datos a través de un socket. Cuando se termina el handshake y la conexión queda establecida, se envían todos los datos del proceso en el mismo formato que se guardan en un archivo, mientras que el daemon remoto los lee también en el mismo formato. Para la estandarización de la escritura y de la lectura se crearon funciones que reciben como parámetro cuál es la función que se utilizará para escribir; esto permite la reutilización de código y además una mejor extensibilidad del programa.

Cuando se termina de recibir la información del proceso en el sistema destino y está cargada la estructura, si hay algún proceso cliente en este sistema que se encuentre esperando por un nuevo proceso, se despierta a través de semáforos y se envía la estructura a través del socket UNIX abierto hacia el cliente. Tanto si se está esperando por un nuevo proceso o no, el daemon guarda toda la información que tiene con un nuevo id de tal manera que el proceso pueda ser restaurado.

Al iniciarse la restauración se tiene entonces cargada la estructura del proceso, ya sea porque se recibió luego de esperar un nuevo proceso o porque se pidieron explícitamente los datos de cierto id local o porque se está leyendo la información desde un archivo. Aunque posiblemente redundante, para simplificación del código, el programa lo que hace es guardar de nuevo toda la estructura hacia un archivo en el formato estándar explicado con las funciones antes mencionadas. Esto se debe a que se ejecutará un nuevo proceso que leerá la información desde este archivo.

Es así que al final del programa de restauración se ejecuta el programa virtualizer, que recibe como parámetro un nombre de archivo que es el que contiene toda la información de un proceso. Este nuevo programa lo primero que solicita es ciertas direcciones y longitudes de memoria para guardar los datos del código y el código que será realmente el restaurador. Primero se escribe en la sección de código, en código de máquina, llamadas a `munmap` para borrar de memoria todas las secciones que no van a hacer utilizadas por el nuevo programa pero que sí utiliza virtualizer. De esta manera, luego se itera sobre toda la lista

de información del proceso escribiendo en cada caso el código y los datos que correspondan.

Si el ítem que se está por restaurar describe el `TLS`, se escribe en código de máquina en la sección de código pedida anteriormente, la llamada de sistema `set_thread_area` con los parámetros descritos en la estructura para obtener finalmente un `TLS` igual al que espera el proceso.

Si el ítem que se quiere restaurar es una parte del VMA, se escriben los datos en la sección de datos y lo que se escribe en código de máquina en la memoria pedida para el código es una llamada de sistema a la función `mmap` para pedir la porción de memoria que el programa espera, y luego las instrucciones para copiar los datos de la sección de datos a la sección solicitada por el `mmap`. De esta manera, cuando todas estas instrucciones terminen de ejecutar, se tendrá un VMA exactamente igual al que el proceso tenía en la máquina origen.

Si el elemento es un descriptor de archivo, se intenta primero abrir directamente el archivo desde la dirección que se tiene; si esto falla, se abre un archivo temporal y se imprime esta situación a la salida estándar para que el usuario pueda saber cómo recuperar el archivo. De esta manera, y utilizando la llamada `fcntl` para cambiar el número del descriptor de archivo, se deja abierto el archivo para que pueda ser utilizado posteriormente.

Por último, cuando el ítem describe los registros se escribe en código de maquina todas las instrucciones necesarias para restaurar los registros, ya sea a través de llamadas de sistema, directamente guardando palabras en los registros o a través del stack, para luego escribir un salto hacia la sección de memoria a la que apunta el registro que es el puntero de la instrucción (`eip` o `rip`).

Finalmente en la sección de código se tienen primero todas las instrucciones para borrar las secciones de memoria utilizadas por el virtualizer (para que no entren en conflicto con las del programa generado), luego se configura el `TLS` si es necesario, para posteriormente crear todas las áreas del VMA del proceso con `mmap`, e inicializar los registros. De esta manera, cuando se realiza el salto, se tiene el código cargado por la parte de VMA y los registros en los valores que se estaban utilizando. Por eso, lo último que realiza el programa virtualizer es saltar hacia la dirección de memoria de la sección de código.

Es importante notar que el stack del virtualizer desaparece con las instrucciones `munmap` y el stack del proceso generado se carga con VMA y se configura su dirección en los registros. Además, también se nota que las bibliotecas se cargan enteramente como si fueran una sección de memoria normal, lo cual puede causar conflictos si son muy diferentes las versiones de los sistemas entre los cuales se está migrando. También se nota que el heap del proceso se mantiene igual que el del virtualizer para facilitar problemas y que solo se agranda para que ocupen el tamaño de los datos necesarios.

5.5. Resultados

Se realizaron las pruebas indicadas en el diseño y en esta sección se introducen los resultados obtenidos con el prototipo propuesto. Estos validan el correcto funcionamiento del programa y prueban su uso en situaciones diversas.

5.5.1. Ambiente de las pruebas

Para realizar las pruebas se preparó una máquina virtual para funcionar como la máquina destino de las operaciones, usando como máquina origen la misma que para el desarrollo. La destino entonces es una máquina común de hoy en día, 1GB de RAM y un procesador dual-core de 2.1GHz, en una arquitectura AMD64. Asimismo, la máquina origen es una computadora con 4GB de RAM, mismo procesador y misma arquitectura. Se levanta el daemon y se usa el cliente en las dos computadoras, de tal manera de realizar la migración en un solo sentido, para realizar aquellas pruebas que son solo locales, en las dos máquinas. Se realizan las pruebas con la versión 3.1 de GNU/Linux y la versión 2.6.32 de GNU/Linux.

Además, para probar la arquitectura de 32 bits, se contó también una computadora con esta arquitectura con 2GB de RAM, con un procesador de 2.1GHz, con la versión de GNU/Linux 3.0, y con una máquina virtual también de 32 bits, con la única diferencia siendo que tiene 512MB de RAM. Las pruebas locales se realizan en los dos sistemas, es decir, en una computadora de 32 bits y en una con una arquitectura de 64 bits.

5.5.2. Pruebas

Se realizaran pruebas con cinco programas distintos de cinco maneras distintas. Se explicará a continuación cuáles son cada una de ellas, tratando de explicar qué se espera probar con ellas y qué comandos se deben utilizar. Las cinco maneras de ejecutar las pruebas son las siguientes:

1. `Solo local.` En esta prueba se trata de comprobar el funcionamiento del programa para hacer checkpoints locales para luego reiniciar un proceso. Lo que se debe hacer es iniciar la prueba y ejecutar el programa procs para que salve el checkpoint en un archivo, para luego ejecutarlo nuevamente para que restaure el proceso desde ese archivo. Los comandos a ejecutar de procs son `save` y `restore`. También se debe ejecutar el daemon procsd y usar los mismos comandos pero con ids locales.

2. `Migración a través de archivos.` Se prueba que los archivos que se guardan puedan ser migrados hacia otra computadora. Para esto, se inicia la prueba y se procede como en el ejemplo anterior, con la única diferencia que el comando `restore` se ejecuta en la máquina destino. Los archivos son enviados de cualquier otra manera.

3. `Migración a través de la red.` Se prueba que el programa es capaz de migrar un proceso a otra máquina sin utilizar archivos. Para esto, primero se ejecuta el daemon procsd en la máquina destino y con el cliente procs, se usa el comando `send` con los parámetros correctos y la prueba iniciada, para enviar el proceso hacia el otro sistema. En el otro sistema se usa procs para listar los ids y restaurar el correcto.

4. `Migración instantánea a través de la red.` Se prueba la capacidad del programa de esperar por un nuevo proceso. Para esto, se realizan los mismos pasos que en el ítem anterior con la salvedad que se debe ejecutar en la máquina destino antes de que se envíe el proceso el comando procs con la opción `receive`.

5. `Migración de contenidos archivos` Se realiza una migración a través de la red enviando el contenido de los archivos, para que sea restaurado sin abrir el mismo desde el sistema de archivos. Se utiliza el comando `send` con la opción `-f` para que guarde el contenido, y el comando `restore` también con esa opción para que restaure los archivos directamente de lo enviado por la red sin chequear su existencia en el sistema de archivos local. Todas las pruebas anteriores, en los programas que utilizan archivos, se realizan con un sistema de archivos compartido.

Los cinco programas que se utilizaron para las pruebas son los descritos a continuación:

1. `Iteración.` Es un programa escrito en C, en donde en la función principal solo se realiza una iteración de tamaño suficientemente grande como para demorar más de un minuto. Se quiere probar con esta prueba el funcionamiento básico del programa.

2. `Iteración con impresión en pantalla.` También escrito en C, donde la iteración tiene un sleep para demorar más tiempo y en donde se imprimen en pantalla el número de iteración por el que se encuentra. Se quiere probar que funcione la utilización de bibliotecas y la impresión en pantalla.

3. `Recursión con impresión en pantalla.` A diferencia del anterior, en lugar de utilizar una iteración, se hacen llamadas recursivas a una función para lograr el mismo propósito. De esta manera, se comprueba el correcto funcionamiento del stack y la restauración de memoria.

4. `Lectura de un archivo.` Lo único que realiza el programa es leer un archivo e imprimir en pantalla cada una de sus líneas.

5. `Lectura y escritura de un archivo.` Realiza lo mismo que el anterior, pero en vez de imprimir en pantalla, escribe cada una de las líneas en otro archivo.

5.5.3. Resultados esperados

Para todas las pruebas, lo que se espera es que cuando se reanude el proceso, sea cual sea el procedimiento, este continúe desde la posición que se quedó y termine correctamente. Esto es fácil de comprobar para las pruebas que tienen impresión en pantalla, pero para la primera, lo que se debe hacer es contar el tiempo que demora el programa en situaciones normales para luego comprobar que realmente demore parecido con la migración.

Por último, para comprobar el funcionamiento de los programas que manejan el sistema de archivos, se tiene en cuenta que se debe seguir leyendo desde la posición que se quedó el programa y además, si se está escribiendo, que esto se realice sin problemas y al igual que si el programa fuera local.

5.5.4. Resultados obtenidos

Los resultadas que se obtuvieron fueron los mismos que los esperados. Se tiene en cuenta que no se pudo realizar con éxito la compilación en 32 bits en la computadora con arquitectura de 64 bits para enviar a la computadora

con arquitectura de 32 bits, debido a que la segunda se quedaba sin memoria rápidamente. Por otro lado, no hubo nada más observable en los resultados ni se pretende graficar lo que se obtuvo, ya que no se están haciendo comparaciones de performance.

6. Resumen y conclusiones

En esta sección se hace un repaso corto de lo visto en este trabajo a partir de las ideas que quedaron por el estudio y el prototipo realizado. Se brindan las características generales de la migración de procesos así como la utilidad del prototipo y se presenta una conclusión final del trabajo realizado a modo de cierre.

6.1. Ejecución del proyecto

El proyecto se realizó de manera directa, es decir, sin utilizar ningún método iterativo ni plan de proyecto complicado. Se cumple entonces con lo planteado en la propuesta del proyecto, desarrollando las actividades de estudio del problema, diseño de una solución e implementación de un prototipo. A diferencia de la propuesta, para facilitar la implementación de la solución al problema se relajan las restricciones de seguridad del sistema construido.

	Mayo	Junio	Julio	Agosto	Setiembre	Octubre	Noviembre	Diciembre
Estudio de conceptos previos	■	■	■			■	■	■
Estudio del estado del arte		■	■	■	■			
Análisis del problema				■	■			
Diseño de la solución					■			
Implementación del prototipo					■	■	■	
Testeo							■	■
Documentación					■	■	■	■

Cuadro 3: Detalle de ejecución del proyecto

Un resumen detallado se plantea en el Cuadro 3. Se puede ver como el estudio del estado del arte tomó una cantidad de tiempo importante mientras que el diseño fue más simple una vez entendido el problema. Por el otro lado, se nota como el estudio de conceptos previos se repite hacia el final del proyecto debido a que, durante la implementación, se plantearon nuevas dificultades y posibilidades que merecían ser estudiadas. El resto de la diagramación es similar a otros proyectos de este estilo en cuanto al tiempo y el orden de las etapas.

Más allá de lo planteado con el estudio de los conceptos previos, se quiere notar que no hubieron complicaciones en cuanto a esta ejecución y que ésta permitió un buen desarrollo del problema sin restricciones de tiempo. El proyecto en sí, entonces no tiene mayores características que valga la pena resaltar en cuanto al uso del tiempo y de los recursos disponibles.

6.2. Acercamiento a la gestión de procesos

La mayor dificultad de este problema fue al principio, donde se tuvo que decidir cómo se puede implementar un sistema de este tipo a nivel de usuario. Por suerte se contó con el código de CryoPID como ayuda para este proceso, pero este código no es solo viejo, sino que es mucho más complicado de lo que debería para funcionar correctamente; esto hizo que fuera difícil la investigación del mismo y el entendimiento de las partes.

Por otro lado, también se tuvieron que investigar muchos conceptos teóricos para entender el funcionamiento de los procesos en GNU/Linux, lo que sirvió para pensar una implementación distinta a CryoPID sobre todo para la restauración de procesos. Por otro lado, fue también complicado entender el uso que

los distintos conceptos podrían tener para el programa debido, a que hay poca documentación sobre la implementación de migración de procesos.

Encontrar la información sobre la migración de procesos también llevó sus complicaciones, debido a que mucha de la información que hay sobre este tema es en base a nuevos sistemas o a problemas empresariales. Por lo tanto, encontrar detalles de implementación o de utilización de herramientas, fue más problemático de lo que se pensó al principio. Por otro lado, hay mucha investigación teórica de este tema, lo que permitió entender el funcionamiento y la utilidad de los programas para la migración de procesos.

Por último, es notable mencionar que escribir el código de máquina y encontrar el funcionamiento de las arquitecturas, sobre todo durante el debugging, permitió aprender mucho sobre el sistema y sobre herramientas como gdb y strace. Además, junto a estas funcionalidades, las mejoras que contiene el código en C para evitar errores y para la reutilización del código, también tuvieron sus complicaciones, pero al mismo tiempo sirvieron para mejorar el código.

6.3. Posibles mejoras

La primer mejora importante que debe tener este programa es el control de errores. Debido a que es un prototipo, se dejaron de lado muchos de los problemas que pueden suceder cuando se intenta migrar un proceso. Por ejemplo, si falla una migración inmediata y el proceso original se mata, no existe ninguna forma de reanudarlo, por lo que al utilizarlo hay que tener mucho cuidado.

También se cree que hay muchas partes que se podrían optimizar, tanto de código como de decisiones de implementación. Por ejemplo, si se tiene una discusión entre los servidores antes de guardar el proceso, se puede saber cuáles bibliotecas es mejor no enviar para que se levanten localmente; lo mismo puede suceder con archivos y otras áreas de memoria o datos del programa.

Otra área del programa que se debe mejorar es la interfaz administrativa, para ser más explícito con los errores que se muestran al usuario y además para implementar el modo verboso del programa, ya que la opción existe pero no se imprime información valiosa. Es así que, por ejemplo, cuando ocurren errores no se explica bien cuál es la forma de solucionarlos, o que los comandos que pueden comunicarse con el servidor fallan, aunque no se tengan que comunicar con el mismo.

Es importante también especificar cómo una mejora posiblemente necesaria para el prototipo entregado es poder mantener el proxy de archivos para restaurar todos los descriptores de archivos y que el proceso remoto pueda seguir ejecutando. Esto se debe a que para que sea de gran utilidad este programa, tiene que poder abarcar la mayoría de los programas existentes, así que además se deberían también agregar el proxy de sockets y la restauración de los manejadores de señales.

Por último, es notable que todas las opciones del diseño que no fueron implementadas son importantes para tener un sistema que realmente cumpla con los objetivos planteados anteriormente. Sin embargo, ya se tiene un sistema que cumple con lo más básico de ese sistema y por lo tanto, es mejor dejar estas opciones como posibles mejoras.

6.4. Funcionalidad y posible utilización

La posible utilidad de este programa es muy parecida a la que tiene CryoPID y apunta a un público que, aunque entiende menos del sistema operativo, tiene la necesidad de realizar checkpoints de sus procesos o migrarlos a otra máquina si es necesario; por ejemplo, esto se ve mucho en los ambientes científicos donde procesos que demoran mucho necesitan tener la seguridad de terminar. Por el otro lado, gracias a Internet, se tienen muchos usuarios que también realizan computaciones grandes y que necesitan un sistema que les permita confiar que siempre se contará con un resultado a pesar de un error inesperado.

Sin embargo, como este programa ya utiliza dentro del mismo la migración de proceso y el checkpoint en memoria en un servidor, puede ser muy útil para tener un servidor local donde guardar todos los procesos que se quieran restaurar en algún momento. Además de también contar con la posibilidad de migrar instantáneamente un proceso hacia otra computadora cuando se tienen problemas.

Si se extiende el prototipo, como se piensa hacer para cumplir con todas las funcionalidades de CryoPID, este programa cumplirá las mismas funciones que aquel, además de tener más utilidades y casos de uso debido a su integración de la migración. Se piensa que de esta manera, se puede tener un programa actual y seguro para realizar migraciones de procesos apuntadas a un nivel académico menor que las otras herramientas.

6.5. Trabajos futuros

El código que aquí se implementó debe ser liberado con una licencia compatible con el Free Software, debido a la utilización de código de CryoPID, por lo que se espera que pueda ser utilizado para otras investigaciones o para ser mejorado. De esta manera, se supone que este código puede conformar entonces parte de un sistema más grande y ser mejorado por una misma idea por la que se mantuvo CryoPID.

Además, se espera poder en el futuro implementar las funcionalidades que faltan del programa, ya que se cree que la utilidad del mismo no es cumplida por ningún otro programa actual. Se cree que al mejorar este programa y expandir sus funcionalidades hasta por fuera de aquellas descritas en el diseño, puede ser de gran utilidad para la comunidad del Free Software.

Con el advenimiento de los celulares y sobre todo el sistema operativo Android, se piensa que será posible crear un port de este sistema hacia ese sistema operativo que también está basado en Linux. De esta manera, se podría brindar por primera vez un sistema de migración de procesos en dispositivos móviles, algo que claramente va a ser mucho más estudiado en el futuro cuando estos dispositivos sean mejores y se encuentren con todavía más uso.

Por último, es importante notar que se espera dejar el programa en un estado un poco más estable para poder ser útil en las posibles ideas que se mencionan en la siguiente sección. De esta manera, la misma comunidad que usaba CryoPID puede empezar a usar este programa que pretende ser una versión más fácil de usar y más nueva. Se espera también conseguir nuevas ideas para poder observar la migración de procesos y su uso en el futuro.

6.6. Conclusión

Para finalizar este trabajo se quiere explicar cómo este trabajo ha brindado una mirada global al problema de la migración de procesos y a sus soluciones existentes. Se han presentado los principales sistemas, cada uno con sus ventajas y problemas, y brindando una idea general de su utilización y objetivos. De esta manera, se piensa que los sistemas de migración de procesos son un tema muy estudiado teóricamente, pero con pocas aplicaciones prácticas. Se ha explicado cómo se podría brindar una mejor solución y se ha implementado un sistema que demuestra la posibilidad y la usabilidad de un sistema de este estilo.

Se espera que con este trabajo se pueda difundir más la migración de procesos y sirva como referencia para futuros estudios que necesiten tener a disposición la mayoría de los problemas y soluciones que existen en los distintos sistemas. Además, con la mirada global que se ha establecido, se tiene una buena base para realizar trabajos futuros e investigaciones sobre cómo lograr que los sistemas existentes, o algún sistema nuevo, pueda alcanzar su mejor desarrollo y una mejor base de usuarios.

Con el prototipo y el diseño implementados, se ha establecido la posibilidad de uso de estos sistemas y lo importantes que pueden llegar a hacer en un futuro, sobre todo cuando se tenga más necesidad de computación y más velocidades en las redes. También se ha explicado lo suficiente cómo la transferencia de la computación a la nube puede ser implementada con varios sistemas de migración de procesos que permitan redistribuir y aprovechar todos los recursos existentes que hoy en día se mantienen desaprovechados.

Aunque no se brinda la mejor solución como implementación, se brinda una posibilidad y una base para el concepto teórico de la migración de procesos que permite entender la idea y mejorar sus principales aspectos. Además, estos sistemas van a ser cada vez más importantes y van a permitir una nube global y distribuida que permita que la computación se desarrolle de manera paralela y con el balanceo más eficiente para lograr los cometidos tanto de la ciencia como de los usuarios comunes. No se pretende que el prototipo solucione todos estos problemas, pero se quiere que sirva como idea básica para la implementación de futuras soluciones y que invite a crear mejores conceptos prácticos para la migración de procesos.

El futuro pretende brindar muchas funcionalidades y entre ellas se encuentra la distribución de la computación, y este concepto es algo que solo puede ser desarrollado en tanto la migración de procesos permita un mejor aprovechamiento de los recursos. Es así que se espera que los sistemas móviles permitan tener la misma computación que una computadora de escritorio y al mismo tiempo tener las mismas posibilidades, para así siempre tener los procesos necesarios en cada uno de los dispositivos que usamos. Es necesario que los conceptos detrás de este trabajo y las implementaciones brindadas sean mejoradas para que cuando las redes y las máquinas lo permitan, se tenga preparado un sistema que pueda distribuir el procesamiento y el conocimiento de tal manera que la nube, y el poder que esta brinda, se distribuya entre todos los sistemas que estén conectados a una red.

Referencias

[1] D. S. Milojíčić, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou, "Process migration," *ACM Comput. Surv.*, vol. 32, pp. 241–299, September 2000. [Online]. Available: http://doi.acm.org/10.1145/367701.367728

[2] G. Andrews, *Foundations of Multithreaded, Parallel, and Distributed Programming.* Addison-Wesley, 1999.

[3] R. Schollmeier, "[16] a definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications," in *Proceedings of the First International Conference on Peer-to-Peer Computing*, ser. P2P '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 101–. [Online]. Available: http://dl.acm.org/citation.cfm?id=882470.883282

[4] S. Thulasidasan, "Issues in process migration," On Line. public.lanl.gov/sunil/pubs/csci555tp.pdf. Last accesed: 23 nov 2011, 2000.

[5] P. Dasgupta, "General purpose process migration," On Line. http://cactus.eas.asu.edu/partha/Papers-PDF/1900-2001/proc-migration.pdf. Last Accesed: 23 nov 2011.

[6] G. Vallee, R. Lottiaux, D. Margery, and C. Morin, "Ghost process: a sound basis to implement process duplication, migration and checkpoint/restart in linux clusters," in *Proceedings of the The 4th International Symposium on Parallel and Distributed Computing.* Washington, DC, USA: IEEE Computer Society, 2005, pp. 97–104. [Online]. Available: http://dl.acm.org/citation.cfm?id=1126254.1126682

[7] X. Wang, Z. Zhu, Z. Du, and S. Li, "Multi-cluster load balancing based on process migration," in *Advanced Parallel Processing Technologies*, ser. Lecture Notes in Computer Science, M. Xu, Y. Zhan, J. Cao, and Y. Liu, Eds. Springer Berlin / Heidelberg, 2007, vol. 4847, pp. 100–110. [Online]. Available: http://dx.doi.org/10.1007/9783540768371_14

[8] "Putting PANTS on Linux," On Line. http://web.cs.wpi.edu/ claypool/mqp/pants/. Last accessed: 5 sep 2011.

[9] R. Buyya, T. Cortes, and H. Jin, "Single system image," *Int. J. High Perform. Comput. Appl.*, vol. 15, pp. 124–135, May 2001. [Online]. Available: http://dl.acm.org/citation.cfm?id=1080643.1080652

[10] D. Bovet and M. Cesati, *Understanding the Linux Kernel, Second Edition*, 2nd ed., A. Oram, Ed. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2002.

[11] N. Vasudevan and P. Venkatesh, "Design and implementation of a process migration system for the linux environment," 3rd International Conference on Neural, Parallel and Scientific Computations, 2006.

[12] J. M. Smith, "A survey of process migration mechanisms," *SIGOPS Oper. Syst. Rev.*, vol. 22, pp. 28–40, July 1988. [Online]. Available: http://doi.acm.org/10.1145/47671.47673

[13] P. E. Chung, Y. Huang, S. Yajnik, G. Fowler, K.-P. Vo, and Y.-M. Wang, "Checkpointing in cosmic: A user-level process migration environment," in *Proceedings of the 1997 Pacific Rim International Symposium on Fault-Tolerant Systems*, ser. PRFTS '97. Washington, DC, USA: IEEE Computer Society, 1997, pp. 187–. [Online]. Available: http://dl.acm.org/citation.cfm?id=826040.827027

[14] G. Atrio, M. Benzo, and G. Utrera, "Taller v: Checkpoint y migración de procesos en ambiente distribuido," Proyecto de grado, Facultad de Ingeniería, Universidad de la República, 1996.

[15] S. Microsystems, "Nfs: Network file system protocol specification," United States, 1989.

[16] B. Miller and D. Presotto, "Xos: an operating system for the x-tree architecture," *SIGOPS Oper. Syst. Rev.*, vol. 15, pp. 21–32, April 1981. [Online]. Available: http://doi.acm.org/10.1145/1041459.1041462

[17] M. L. Powell and B. P. Miller, "Process migration in demos/mp," Berkeley, CA, USA, Tech. Rep., 1983.

[18] A. Barak, A. Braverman, I. Gilderman, and O. Laden, "Performance of pvm with the mosix preemptive process migration scheme," in *Proceedings of the 7th Israeli Conference on Computer-Based Systems and Software Engineering*. Washington, DC, USA: IEEE Computer Society, 1996, pp. 38–. [Online]. Available: http://dl.acm.org/citation.cfm?id=857173.857265

[19] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel, "The locus distributed operating system," *SIGOPS Oper. Syst. Rev.*, vol. 17, pp. 49–70, October 1983. [Online]. Available: http://doi.acm.org/10.1145/773379.806615

[20] "Inferno (operating system)," On Line. http://en.wikipedia.org/wiki/Inferno_(operating_system). Last accesed: 9 sep 2011.

[21] "Inferno-os - inferno distributed operating system - google project hosting," On Line. http://code.google.com/p/inferno-os/. Last accesed: 9 sep 2011.

[22] "Condor," On Line. http://www.cs.wisc.edu/condor/. Last accessed: 5 sep 2011.

[23] D. Thain, T. Tannenbaum, and M. Livny, "Distributed computing in practice: the condor experience: Research articles," *Concurr. Comput. : Pract. Exper.*, vol. 17, pp. 323–356, February 2005. [Online]. Available: http://dl.acm.org/citation.cfm?id=1064323.1064336

[24] M. Litzkow, M. Livny, and M. Mutka, "Condor - a hunter of idle workstations," 8th International Conference on Distributed Computing Systems, 1988.

[25] "Platform LSF," On Line. http://www.platform.com/workload-management/high-performance-computing. Last accessed: 5 sep 2011.

[26] S. Zhou, X. Zheng, J. Wang, and P. Delisle, "Utopia: a load sharing facility for large, heterogeneous distributed computer systems," *Softw. Pract. Exper.*, vol. 23, pp. 1305–1336, December 1993. [Online]. Available: http://dl.acm.org/citation.cfm?id=173497.173499

[27] "Tivoli Workload Scheduler Loadleveler," On Line. http://www.ibm.com/systems/software/loadleveler/. Last accessed: 5 sep 2011.

[28] K. Dickinson, C. Homic, and B. Villamin, "Putting pants on linux: Transparent load sharing in a beowulf cluster," On Line. http://web.cs.wpi.edu/ claypool/mqp/pants/report.pdf. Last accesed: 23 nov 2011, 2000.

[29] J. S. Plank, M. Beck, G. Kingsley, and K. Li, "Libckpt: Transparent checkpointing under unix," Knoxville, TN, USA, Tech. Rep., 1994.

[30] "Eduardo Pinheiro Checkpoint Project," On Line. http://www.research.rutgers.edu/ edpin/epckpt/. Last accessed: 5 sep 2011.

[31] "Linuxpmi," On Line. http://linuxpmi.org/trac/. Last accesed: 9 sep 2011.

[32] "Dmtcp: Distributed multithreaded checkpointing," On Line. http://dmtcp.sourceforge.net/. Last accesed: 9 sep 2011.

[33] "Cryopid - a process freezer for linux," On Line. http://cryopid.berlios.de/. Last accesed: 9 sep 2011.

[34] "Boinc," On Line. http://boinc.berkeley.edu/. Last accesed: 9 sep 2011.

[35] D. P. Anderson, C. Christensen, and B. Allen, "Designing a runtime system for volunteer computing," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, ser. SC '06. New York, NY, USA: ACM, 2006. [Online]. Available: http://doi.acm.org/10.1145/1188455.1188586

[36] "Et, phone seti@home!" On Line. http://science.nasa.gov/science-news/science-at-nasa/1999/ast23may99_1/. Last Accesed: 23 nov 2011.

[37] "Virtualbox teleporting (the art of virtualization)," On Line. http://blogs.oracle.com/vreality/entry/teleporting. Last Accesed: 23 nov 2011.

[38] "Vmware vmotion for live migration of virtual machines," On Line. http://www.vmware.com/products/vmotion/overview.html. Last Accesed: 23 nov 2011.

[39] "Samsung and vmware bringing virtualization to android," On Line. http://mobile.slashdot.org/story/11/09/07/1644206/Samsung-and-VMWare-Bringing-Virtualization-to-Android. Last accesed: 11 sep 2011.

[40] "What you capture is what you get: A new way for task migration across devices," On Line. http://googleresearch.blogspot.com/2011/07/what-you-capture-is-what-you-get-new.html. Last accesed: 11 sep 2011.

[41] "Securityteam/roadmap/kernelhardening - ubuntu wiki," On Line. https://wiki.ubuntu.com/SecurityTeam/Roadmap/KernelHardening#ptrace_Protection. Last accesed: 8 nov 2011.

[42] A. Telephone and T. Company, *System V application binary interface: UNIX System V*, ser. UNIX software operation. Prentice-Hall, 1990. [Online]. Available: http://books.google.com/books?id=mrImAAAAMAAJ

Apéndice A. Páginas `man` de los programas

Se presentan en esta apéndice las páginas `man` de los programas del sistema `procs` para tener una vista rápida tanto de las capacidades del mismo como una ayuda sobre su uso.

A.1. `procs`

NOMBRE

procs - cliente de procs para migrar y hacer checkpoints de procesos

SINOPSIS

procs [save | restore | send | receive | list | delete] [-h | -s | -n | -o | -K | -f | -v]

DESCRIPCIÓN

procs es capaz de guardar el estado de un proceso a un archivo y restaurarlo del mismo, migrar un proceso instantáneamente a otra computadora o guardar el estado del mismo en el servidor local. Permite tener archivos de respaldo (o copias en el servidor local) por si un programa falla. Si el proceso utiliza algo distinto a la memoria, los registros, archivos abiertos y a los descriptores de archivo estándar podrá ser migrado pero puede no funcionar correctamente una vez restaurado.

OPCIONES

-h –help
muestra la ayuda del programa

-v –verbose
salida verbosa

save [PID]
guarda el estado del programa apunta por PID o NAME.

-n –name
nombre del proceso para la migración

-o –output
archivo para guardar el estado

-K –nokill
mantener el proceso en ejecución luego de la migración

- f –file-save
guardar toda el contenido de los archivos abiertos

restore [FILE|PROCS_ID]
restaura el proceso guardado en el archivo FILE. Si no existe el archivo, asume un PROCS_ID del servidor local.

send [FILE|PROCS_ID|PID]
envía el proceso referenciado por PID, FILE o PROCS_ID (verificando en es orden) al servidor especificado.

-s –server
IP:PORT o HOSTNAME:PORT del servidor al cual se enviará el proceso.

-n –name
nombre del proceso al que se le debe guardar el estado

-K –nokill

mantener el proceso en ejecución luego de la migración

receive [PROCS_ID]

restaura el programa referencia por PROCS_ID del servidor local o espera hasta que el próximo proceso arribe al servidor local y lo ejecuta.

list

muestra los PROCS_ID del servidor local.

delete [PROCS_ID]

borra el proceso referenciado por PROCS_ID del servidor local.

EJEMPLOS

procs save 15325 -o tmp

guardar proceso con PID al archivo tmp

procs restore tmp

restaurar procesos guardado en el archivo tmp

procs receive

esperar al próximo proceso que llegue al servidor local y ejecutarlo

procs send -n test -s 192.56.1.101

enviar proceso con nombre test al servidor procsd(1) en 192.56.1.101 con el puerto estándar

A.2. procsd

NOMRBE

procsd - servidor de procs para recibir y guardar procesos

SINOPSIS

procsd [HOSTNAME [:PORT]] [-h | -f | -v]

DESCRIPCIÓN

procsd es un daemon para guardar y migrar procesos. Funciona con procs(1) como su interfaz. Guarda cada proceso que recibe con un identificador único que luego puede ser utilizado por el ciente para restaurarlo.

OPCIONES

-h –help

muestra la ayuda del programa

-v –verbose

salida verbose

-f –nofork

no hacer fork ni daemonizar el programa

EJEMPLOS

procsd 192.168.56.1:6065 -f

correr el servidor sin daemonizar con IP 192.168.56.1 y puerto 6065

procsd

correr el proceso con el IP y puerto por defecto(127.0.0.1:12092)

Apéndice B. Guía rápida de uso

En este apéndice se pretende brindar un guía rápida para poder usar el sistema `procs` especificado en esta trabajo. Primero que nada es importante entender que el sistema esta compuesto por dos programas: `procsd` y `procs`. El primero se encarga de actuar como servidor para recibir procesos, por lo que es necesario que esté siendo ejecutado en la máquina destino para realizar una migración. El segundo actúa tanto de cliente para un programa `procsd` local como para uno remoto.

Por lo tanto, lo primero que se debe realizar para iniciar una migración es ejecutar desde la línea de comandos `procsd IP:PORT`, donde IP y PORT son los valores con los que trabajará el servidor. Este programa se encarga entonces de abrir el puerto necesario y escuchar las llamadas que se realicen para recibir procesos, además de también abrir un `socket UNIX` en el archivo `/etc/procs/procs.sock` (por lo se debe crear y dar permisos al usuario) para recibir pedidos localmente.

Luego de que este proceso se encuentra en ejecución en la máquina destino, en la máquina remota se debe ejecutar el comando `procs send PID -s IP:PORT`, que realizará todo lo necesario para guardar los datos del procesos referenciado por PID y enviarlos al servidor `procsd` que se encuentra en la dirección IP especificada. Si tan solo se quiere guardar el proceso a un archivo, no es necesario que se esté ejecutando el proceso `procsd`, y se ejecuta el siguiente comando `procs save PID -o ARCHIVO`, donde ARCHIVO es el nombre del archivo donde se guardará el estado del proceso.

Cuando se quiere restaurar el procesos, se necesita primero obtener el identificador que el servidor local le ha dado al proceso recibido. Para esto se ejecuta el comando `procs list` que se conecta con el proceso `procsd` local y muestra en pantalla todos los identificadores de procesos que se han guardado. Una vez obtenido el identificador, se ejecuta el comando `procs receive PROCID` donde PROCID es el identificador del proceso en el servidor local. Es importante mencionar que si se quiere borrar un proceso del servidor local, se puede ejecutar `procs delete PROCID`.

Si el proceso se ha guardado a un archivo, con el siguiente comando se reanuda el mismo: `procs restore ARCHIVO`, donde ARCHIVO es el nombre del archivo que contiene el estado del proceso guardado con el comando anteriormente explicado. Es importante notar que si el archivo no se encuentra, se intentará conectar con el servidor `procsd` local para utilizar ARCHIVO como un identificador. Por último, para restaurar también se puede ejecutar el comando `procs receive` sin ningún parámetro, que lo que hace es conectarse con el servidor local para esperar a que llegue una nueva migración de un proceso para reanudarlo.

Cuando el proceso contiene archivos abiertos, el sistema supone que se tiene un sistema de archivos compartido entre las dos computadoras con una misma ruta absoluta. Si esto no es así, y al reanudar el proceso no se encuentra el archivo buscado, se creará un archivo temporal en `/tmp`, y se indicará en el error estándar el nombre del archivo original y la ruta absoluta del archivo recién creado. De esta manera, el proceso puede seguir escribiendo y el usuario sería el encargado de unir los archivos que el proceso ha escrito.

Para evitar la necesidad de tener un sistema de archivos compartidos se le puede indicar tanto al comando `procs send` como al comando `procs save` la

bandera -f, que indicará que se debe guardar todo el contenido de los archivos abiertos. De esta manera, toda esta información se enviará a la computadora destino que restaurará los archivos en `/tmp` indicándole al usuario el nombre del archivo original y la ruta en la que fue restaurado, para que éste pueda luego encargarse de restaurarlos en la máquina origen si es necesario.

A través de estos comandos y sus distintas opciones, se puede obtener una migración de procesos entre computadoras o un sistema de checkpointing para tener archivos de respaldos de distintos procesos.

Apéndice C. Guía para desarrolladores

En este apéndice se quiere brindar una guía del sistema para su posterior extensión y como una pequeña documentación de lo realizado. Se entiende que así se puede lograr que otros desarrolladores puedan mejorar o entender el sistema `procs`.

C.1. Módulos

El sistema está compuesto por 7 módulos distintos: `balancer`, `loader`, `procs`, `saver`, `sender`, `virtualizer` y `utils/common`. Entre estos siete se reparten todas las funciones necesarias para la migración, el checkpointing, el programa cliente y el daemon. Cada uno tiene sus responsabilidades como se indica en el diseño del mismo y mantienen una separación de necesidades para evitar problemas.

El módulo `utils/common` merece ser mencionado primero puesto que en él se encuentran todas las funciones y estructuras compartidas por los demás. Primero que nada se tienen definidos todos los `struct` de C que definen a los procesos y a sus partes. Además se tiene también una pequeña implementación de una lista doblemente encadenada que se utiliza tanto en la estructura del proceso como para guardar los identificadores del servidor.

También se tienen en éste módulo todas las funciones que son relativas a las estructuras definidas en el mismo, por lo que se tiene una función para leer procesos tanto sea de un archivo como de un `socket`, y para poder modificar las listas. Por último, se tienen además, algunas funciones auxiliares que son utilizadas por varios módulos.

En el módulo `balancer` se contiene toda la lógica necesaria para el daemon. Por lo tanto, en el mismo están definidas todas las funciones necesarias para abrir puertos de Internet y para abrir `sockets UNIX` para poder comunicarse tanto remota como localmente. Además contiene las estructuras para poder guardar un proceso en una lista y las funciones para agregar o remover un identificador (y por lo tanto sus datos).

El módulo `loader` contiene la función necesaria para restaurar un procesos y poder levantar el `virtualizer` que será el encargado de cargar todos los datos del proceso y ponerlo en ejecución. Por lo tanto este módulo es más simple que otros. Asimismo, se tiene el módulo `sender` que sólo contiene una función que se encarga de enviar un proceso guardado hacia otro sistema.

Por otro lado, el `saver` se encarga de toda la parte del sistema relativa a parar un proceso, obtener toda su información y reanudarlo o matarlo. Este módulo recibe el identificador del proceso a guardar y se encarga de obtener su memoria, registros y archivos abiertos. Todos estos datos son luego guardados en una estructura de proceso que será devuelta para luego poder ser guardada a un archivo o enviada a otra computadora.

El `virtualizer` contiene todo el código necesario para, a partir de un nombre de archivo que será leído, obtener toda la información de un proceso que será cargada en sí mismo para poder luego reanudar su ejecución. Así, se encarga de abrir todos los archivos y de restaurar toda la memoria para que, cuando se restauren los registros, se tenga todo listo para continuar con la ejecución del proceso migrado.

Por último, el módulo `procs` contiene la lógica del cliente del programa y realiza todas las llamadas necesarias hacia los otros módulos o hacia el `procsd` local, para cumplir con el requerimiento del usuario.

C.2. Comunicación

La comunicación del sistema se realiza tanto a través de un archivo de `socket UNIX` en la ruta absoluta `/etc/procs/procs.sock` (por lo que esta ruta debe ser creada y se le debe dar permisos al usuario para utilizarla) como a través de `sockets` de Internet para la comunicación remota. Más allá de estos datos, no es necesaria una explicación más extensa de la comunicación ya que toda la escritura o lectura de datos se hace de manera simple o a través de la funciones que provee el módulo `utils/common`.

C.3. Funciones

En esta sección se explicarán brevemente las funciones más importantes del sistema de manera de poder entender su funcionamiento. Cada función se encuentra en el módulo que le corresponde según el problema que soluciona.

`receive_process` es la función encargada de recibir un proceso que ha sido enviado a través de la red. Para esto utiliza la función de lectura de `utils/common` que le permite obtener toda la información del estado del proceso que se le irá enviando a través de la red. Luego de obtenida la estructura del proceso, se agrega a la lista de identificadores para poder ser restaurado y se despierta a cualquier cliente que esté esperando por el próximo proceso para restaurar (con la opción `procs receive`).

`admin` se encarga de procesar los pedidos de un cliente a un servidor `procsd` local. De esta manera, según la opción que se le pase, tiene la posibilidad de enviar un proceso al cliente para que este lo restaure, de listar todos los identificadores de procesos o de borrar uno de estos.

`main`**(balancer)** es la función principal del programa `procsd` y por lo tanto, se encarga de obtener todas las opciones que se le pasaron al programa, inicializar los `sockets` tanto para llamadas remotas como para llamadas locales y de crear el daemon que se ejecutará para esperar nuevos procesos a migrar.

`restore_process` es la única función del módulo `loader` y se encarga, dada una estructura de datos de un proceso, escribirla en un archivo a través de las funciones estandarizadas en `utils/common`, para luego ejecutar el programa `virtualizer` con el nombre de archivo creado que se encargará de leer la estructura del proceso y reanudarlo.

`main`**(procs)** es el encargado de obtener todas las opciones del cliente y pasárselas a los demás módulos para realizar lo requerido por el cliente. Además, también realiza llamadas a otras funciones dentro del mismo módulo para comunicarse con el servidor `procsd` local.

`save_process` guarda toda la información de un proceso. Para esto, primero utiliza `ptrace` para dejar de ejecutar el proceso a migrar y llama a distintas funciones para obtener las distintas partes. Es importante el orden en que obtiene las partes, primero el `TLS`, luego el `VMA`, los archivos abiertos y por último los registros, puesto que así es cómo cada una va a ser restaurada.

`fetch_chunk_tls` obtiene, a través de `ptrace` y la memoria del proceso, el `TLS` del proceso en ejecución si es que el sistema está siendo ejecutado en 32 bits. De esta manera logra guardar la información en la estructura del proceso.

`fetch_chunk_regs` obtiene, a través de `ptrace` y el `user_struct` del proceso, todos los valores de los registros del mismo para luego poder agregarlos a la estructura del proceso.

`get_one_vma` es la función encargada de obtener toda la información de una línea del archivo `maps`. Se realiza una lectura según cómo está creada esta línea y se guardan todos los valores en la estructura del proceso. Utiliza `ptrace` para leer la memoria del proceso que será migrado.

`fetch_chunk_vma` lee el archivo `maps` para poder llamar a la función anterior línea por línea e ir llenando la estructura del proceso.

`fetch_chunk_fd` utiliza el directorio `fd` y el directorio `fdinfo`, los dos en `procfs`, para obtener todos los archivos abiertos y su información. Si es necesario, lee el contenido de los archivos. Guarda todos estos datos en la estructura del proceso.

`send_process` es la función del módulo `sender` que a través de una dirección IP y un puerto, y la estructura del proceso a ser migrado, se comunica con el `procsd` remoto para enviarle todos los datos.

`main(virtualizer)` es la función principal del módulo `virtualizer` y se encarga de leer de un archivo toda la estructura de un proceso y de pedir la memoria necesaria para realizar las llamadas a otras funciones del módulo que se encargarán de cargar toda la información en el proceso mismo.

`restore_chunk_fd` recibe la información de los archivos abiertos y trata de abrirlos si es necesario. Si no se pueden abrir o se tiene prendida la bandera para no asumir un sistema de archivos compartidos, guarda todo el contenido de los archivos que recibe en archivos temporales, avisándole al usuario. Luego abre descriptores de archivos para que el proceso pueda usar, y les cambia el número de descriptor para que coincida con el del proceso original. Estos descriptores no se cierran para que el proceso restaurado los siga usando.

`munmap_all` es la función encargada de escribir en la memoria de código del proceso de `virtualizer` la lógica necesaria para borrar toda la información de este. De esta manera, cuando el proceso salte a la dirección de memoria del código, se borrarán todas las áreas de memoria no necesarias.

`restore_chunk_vma` escribe en la sección de código de este módulo las llamadas al sistema tanto para mapear las áreas de memoria como para luego copiar los datos a estas áreas.

`restore_chunk_tls` escribe también en la sección de código una llamada al sistema para modificar el `TLS` del proceso según lo que necesite el proceso restaurado.

`restore_chunk_regs` es la función final de este proceso y se encarga tanto de escribir en la sección de código todas las instrucciones necesarias para que se restauren los valores de los registros, como la llamada final para saltar a la posición en la que se encontraba el proceso restaurado, luego de que todos los datos hayan sido cargados correctamente.

`write_process` se encarga de escribir una estructura de un proceso a través de una función que se le pasa como parámetro tanto sea en un archivo como en un `socket`. De esta manera queda estandarizada la manera de escribir la estructura. Se realiza una iteración sobre todas las partes de un proceso y cada una se escribe de la manera que corresponda según sus datos.

`restore_process` es una función parecida a la anterior pero que se encarga de la lectura de la estructura de un proceso y también intenta estandarizar las comunicaciones.