УДК 004.415

В. А. Гилев

Новосибирский государственный университет
ул. Пирогова, 2, Новосибирск, 630090, Россия
E-mail: adarik@gmail.com

НОВЫЙ ПОДХОД К ПЕРЕНОСУ ПРОЦЕССОВ В LINUX-СИСТЕМАХ

Базовые понятия

Развитие современных информационных технологий, широкое распространение распределенных и кластерных систем, а также обилие как вычислительных задач, так и пользовательских сервисов, запускаемых на них, привели к необходимости создания механизмов балансировки загрузки.

Балансировку загрузки здесь можно определить как специализированный механизм, собирающий информацию о загруженности узлов распределенной или кластерной компьютерной системы и на основе полученной информации переносящий подзадачи или процессы с одного, более загруженного, узла на другой, менее загруженный. Это позволяет значительно ускорить выполняемые в системе операции, снять чрезмерную нагрузку на сеть в случае задач, основанных на взаимодействии с внешней сетью, а также снять риск отказа системы из-за чрезмерной перегруженности одних узлов и простоя других. Балансировка загрузки имеет три составляющих: механизм сбора информации о загрузке узла, в том числе о степени загрузки, приходящейся на долю каждого процесса, и соединениях каждого процесса; механизм анализа и принятия решений о перераспределении процессов между узлами и механизм переноса процессов.

Под *переносом процессов* здесь подразумевается следующее: прекращение выполнения процесса на одном узле системы; сохранение его состояния, в том числе информации об используемых ресурсах, памяти, регистрах процессора, используемых файлах и связях с другими процессами; перенос этих данных на другой узел и, наконец, на основе этих данных создание нового процесса на новом узле, идентичного ранее выполняемому, будто прежний и не прекращал выполняться.

В силу того, что большинство существующих кластерных и распределенных систем построены на основе операционной системы Linux и ее модификаций, а также ввиду открытости ее исходного кода большое количество существующих механизмов переноса процессов работают именно с процессами Linux. И в дальнейшем под *переносом процессов* будет подразумеваться именно *перенос процессов* Linux.

Помимо использования *переноса* для балансировки загрузки, системными администраторами Linux-сетей был найден широкий круг задач, в котором этот механизм нашел свое применение. К ним относятся: обеспечение отказоустойчивости системы путем *переноса процессов* с неисправных узлов на исправные; локализация доступа к данным путем *переноса процессов* ближе к используемым им данным; ускорение времени отклика системы путем *переноса процессов* ближе к пользователям; улучшение доступности служб и администрирования путем *переноса процессов* на другой узел перед обслуживанием узла, на котором они были запущены, так, чтобы приложения могли продолжить свое выполнение с минимальным временем простоя, и другие.

Обоснование необходимости нового подхода

Среди основных требований, предъявляемых к переносу процессов, можно отметить следующие: прежде всего, перенос образа процесса, состояния регистров, открытых файлов,

ISSN 1818-7900. Вестник НГУ. Серия: Информационные технологии. 2007. Том 5, выпуск 1
© В. А. Гилев, 2007

уникальных идентификационных номеров (UID и GID), текущей рабочей директории, а также состояния и свойств терминала и обработчиков сигналов. Этих требований было достаточно для использования механизма переноса для трудоемких кластерных вычислений, где задача разбивалась на несколько абсолютно независимых и не взаимодействующих частей. Однако не все задачи могут быть разделены таким образом. Как же быть с интерактивными задачами и теми, для выполнения которых требуется взаимодействие с сетью или с другими процессами, например использующими программные каналы (pipes) или сокеты (sockets)?

Таким образом, возникли еще два дополнительных требования к механизму переноса: возможность переносить процессы, взаимодействующие через программные каналы и сокеты, не нарушая соединения.

Программный канал можно кратко определить как системный интерфейс для передачи данных между процессами в двух направлениях внутри одного узла. И именно то, что все взаимодействие происходит в рамках одного узла, делает перенос процессов, взаимодействующих посредством этого механизма, относительно легким. Это делается посредством переноса обоих взаимодействующих процессов на новый узел и одновременного запуска. Существует несколько реализаций переноса процессов с поддержкой программных каналов.

Сокет представляет собой системный интерфейс для передачи данных через протоколы TCP и UDP как внутри одного узла, так и между разными узлами. В реальных Unix-системах сокеты служат также для других целей, но в рамках этой работы будет рассматриваться именно такое определение. Этот интерфейс ссылается на структуру сокетов в ядре операционной системы Linux, хранящую данные об одном TCP-соединении, в том числе информацию о процессах, связанных с ним, его состоянии, о маршрутизации для его пакетов, очередях TCP, а также о номерах пакетов для поддержания связи. Любое открытое TCP-соединение идентифицируется четырьмя параметрами:

– IP-адрес и порт одной стороны соединения;
– IP-адрес и порт другой стороны соединения.

При переносе процесса вместе с сокетом возникает следующая проблема: каким образом переместить процесс на другой узел и при этом не изменить эти четыре параметра или изменить их на обоих концах соединения так, чтобы установленная связь не была нарушена?

Существует несколько подходов к решению данной проблемы.

Оставлять на месте процесса агент, маршрутизирующий данные соединения. При этом, очевидно, эти четыре параметра никак не затрагиваются. Но для балансировки загрузки такой метод неприменим, в силу того что балансировка предназначена для разгрузки слишком загруженных узлов, а оставление агента не только чрезмерно нагружает сеть в системе, но и создает дополнительную нагрузку в виде самого агента.

Менять параметры соединения удаленной стороны. Эти четыре параметра также останутся согласованными. Однако для балансировки загрузки и этот метод неприменим, в силу того что он значительно сужает круг задач, запускаемых на таких системах. Например, это не позволяет запускать такие распространенные задачи, как сетевые сервисы и приложения, использующие соединения со службами и системами, которые мы не в состоянии контролировать.

Использовать для всех соединений системы единый proxy-сервер, который будет поддерживать соединение и перенаправлять его пакеты в случае переноса процесса. Четыре вышеуказанных параметра также остаются неизменными. Но этот подход создает такие дополнительные трудности, как единственность точки доступа в систему, а значит, и единственность точки, неисправность которой приведет к невозможности сетевого взаимодействия всей системы; невозможность использования переноса в системах с несколькими точками доступа, в частности в системах, сильно разнесенных географически.

Отсюда вытекает необходимость разработки нового подхода к переносу открытых TCP-соединений, не обладающего недостатками вышеуказанных подходов.

Некоторые из существующих реализаций неудобны еще и тем, что меняют код ядра, в том числе исходный код реализации IP-стека и TCP-протокола, что добавляет дополнительную трудность для внедрения переноса процессов в стабильно работающие системы. Ведь для установки такого механизма требуется полная перекомпиляция ядра Linux, а это не всегда приемлемо.

Таким образом, обозначена потребность в разработке механизма переноса процессов, удовлетворяющего следующим требованиям:

– в первую очередь, механизм должен предоставлять возможность переноса образа процесса, состояния регистров, открытых файлов, уникальных идентификационных номеров (UID и GID), текущей рабочей директории, а также состояния и свойств терминала и обработчиков сигналов, а также открытых программных каналов;

– возможность переноса открытых TCP-соединений:

– абсолютно прозрачно для удаленной стороны соединения;

– не оставляя на месте процесса агент, маршрутизирующий данные соединения;

– не используя proxy-сервер для маршрутизации соединений или данных соединений системы;

– должен быть легко интегрируем в уже работающие системы.

Существующие методы решения и их недостатки

Как упоминалось выше, в настоящее время существует три подхода к переносу процессов вместе со всеми их открытыми TCP-соединениями: оставлять на месте процесса агент, маршрутизирующий данные соединения; менять параметры соединения удаленной стороны; использовать для всех соединений системы единый proxy-сервер. Для того чтобы понять, чем же предлагаемое нами решение отличается от них и чем оно принципиально лучше, требуется более детальное их рассмотрение.

Наиболее известной реализацией первого подхода является система MOSIX [Barak et al., 1998], разработанная в Institute of Computer Science of The Hebrew University of Jerusalem. Ее основная идея заключается в следующем: каждый процесс MOSIX имеет свой так называемый «уникальный домашний узел» (Unique Home Node, или UHN), где он был создан. Обычно это узел, на котором авторизовался пользователь. Когда возникает необходимость перенести процесс на другой узел, мигрирующий процесс разделяется на два контекста: контекст пользователя, который мигрирует, и контекст системы, зависящий от UHN и не способный к миграции. Контекст пользователя, называемый также удаленным (remote), содержит программный код, стек, данные, карты памяти и регистры процесса. Удаленный контекст инкапсулирует процесс, когда он запускается на уровне пользователя. Системный контекст, называемый также представителем (deputy), содержит описания ресурсов, которые процесс присоединяет к себе, и стек ядра для запуска системного кода от имени процесса. Представитель инкапсулирует процесс, когда он запускается на уровне ядра. Это сохраняет узло-зависящую часть системного контекста процесса, следовательно, это должно остаться на UHN процесса. В то время как процесс мигрирует в течение времени между различными узлами, представитель никогда не мигрирует. Связь этих двух контекстов реализуется посредством соединительного канала для взаимодействия. На рис. 1 показано взаимодействие двух процессов. На рисунке левый процесс является нормальным процессом ОС Linux, в то время как правый разделен и его удаленная часть мигрировала на другой узел.

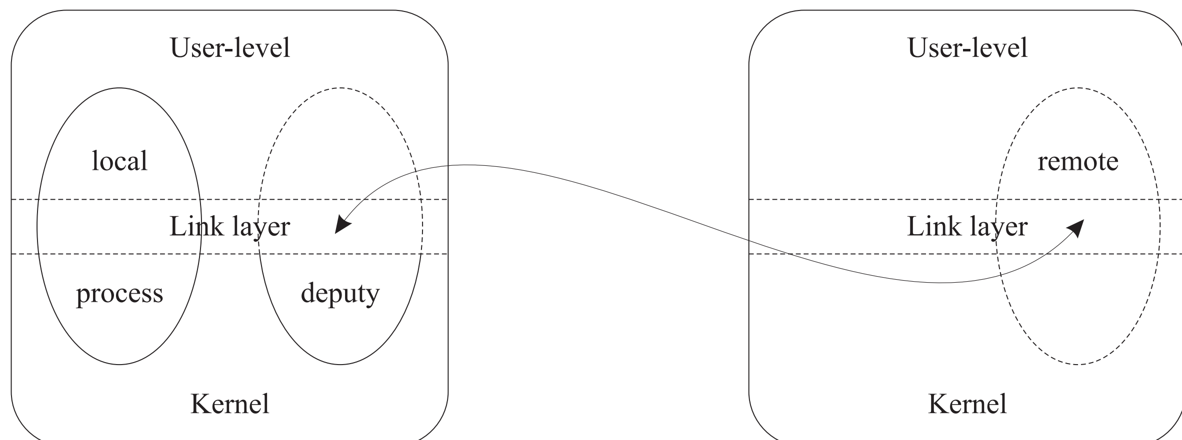

Рис. 1. Взаимодействие процессов. Удаленный и представитель

Системный вызов синхронизируется взаимодействием между двумя контекстами процессов. Все системные вызовы, запущенные процессом, перехватываются на уровне связи (link layer) удаленного узла. Если системный вызов не зависит от узла, то он выполняется локально на удаленном узле. В противном случае системный вызов транслирует представитель, который запускает системный вызов от имени процесса в UHN. Представитель возвращает результат обратно на удаленный узел, который затем продолжает выполнение пользовательского кода.

Из вышеописанного понятно, что все сетевые связи (сокеты) создаются на UHN, таким образом, добавляя накладные расходы, связанные с соединением. Говоря в целом, при переносе открытых соединений, впрочем, как и при создании нового соединения, происходит маршрутизация пакетов с UHN на удаленный узел, что создает не только большую нагрузку на сеть между этими двумя узлами, но и дополнительно нагружает систему в целом за счет дополнительных ресурсов на поддержание подобного разделения процессов.

К недостаткам MOSIX можно также отнести и необходимость перекомпиляции ядра Linux, что обусловливает относительную сложность ее внедрения в существующие системы.

Наиболее успешной и относительно недавней реализацией второго подхода является ZAP [Osman et al., 2002] – механизм переноса простых и сетевых приложений, разработанный в Department of Computer Science of Columbia University. Для переноса открытого соединения в ZAP необходимо, чтобы на обоих концах соединения была установлена данная система. А сам перенос происходит следующим образом: сначала на удаленную сторону отправляется уведомление о приостановке соединения; затем удаленная сторона приостанавливает процесс; после чего мигрирующий процесс останавливается и переезжает на другой узел; восстановившись, он отправляет уведомление о продолжении соединения и о новом расположении конца соединения; затем параметры соединения удаленной стороны меняются в соответствии с полученным уведомлением, и процесс выполняется, продолжив тем самым взаимодействие через данное соединение.

Также механизм ZAP использует третий подход, или подход, основанный на создании единого proxy-сервера, для того чтобы обеспечить возможность взаимодействия с процессами вне системы. Этот факт значительно повышает применимость механизма для решения многих задач. Кроме того, ZAP выполнен в виде модуля ядра, и значит, не требует изменения кода ядра Linux или переустановки системы, что обусловливает легкое внедрение данного механизма в существующие системы.

Тем не менее второй подход, используемый в этой системе, не позволяет решать обширный круг задач, а третий недостаточно безопасен. Комбинация этих двух подходов, конечно, позволяет расширить круг задач, но подвергает систему, его использующую, определенным рискам, что не всегда допустимо.

Кроме ZAP, proxy подход реализует система MSOCKS [Maltz, Bhagwat, 1998] (Carnegie Mellon University и IBM T. J. Watson Research Center). В основе системы лежит довольно простой алгоритм переноса открытого соединения: в начале proxy-серверу отправляется уведомление о переносе процесса на другой узел, после чего proxy прекращает отправлять пакеты процессу в системе, не разрывая при этом связь с удаленной стороной соединения; после этого процесс останавливается и переносится на другой узел; затем, когда процесс восстановился на новом узле, происходит соединение с proxy-сервером и уведомление его о намерении восстановить соединение и текущем расположении процесса; затем proxy просто продолжает перенаправлять пакеты, приходящие с обеих сторон соединения, изменяя их заголовки.

Применяемый подход позволяет использовать перенос процессов и в системах, активно взаимодействующих с внешними ресурсами, контролировать которые мы не в состоянии. Однако и здесь есть свои подводные камни: что произойдет, если proxy-сервер выйдет из строя? Сетевое взаимодействие как внутри системы, так и вне ее будет невозможным, что может привести к неработоспособности всей системы. Это накладывает серьезные ограничения на применимость таких систем. Кроме того, что делать с системами, сильно разнесенными географически, где обращение к proxy-серверу будет создавать огромные задержки?

Таким образом, определяется необходимость в разработке нового подхода (или модификации существующих), который позволил бы избежать всех вышеозначенных проблем.

Предлагаемое решение и его обоснование

Транспортные протоколы TCP и UDP изначально разрабатывались для взаимодействия двух стационарных узлов сети и не предусматривают возможность одного из концов перемещаться с одной машины на другую. В частности, открытое соединение характеризуется, как было сказано выше, четырьмя параметрами, которые не меняются на протяжении всей жизни соединения и идентифицируют соединение среди других. Кратко обозначим их [IP1:port1 – IP2:port2], где IP1, port1 – IP-адрес и порт одной стороны соединения, а IP2, port2 – IP-адрес и порт другой стороны соединения.

При переносе одной стороны на другой узел, с другим IP-адресом и, возможно, с занятым портом port1, возникает несогласованность между идентификационной парой соединения [IP1:port1 – IP2:port2] и его текущей парой [IP3:port3 – IP2:port2]. Здесь IP3, port3 – IP-адрес узла, на который переехал процесс, и порт, используемый для продолжения связи на новом узле. Для разрешения данной несогласованности предлагается следующее решение: каждый процесс при запуске привязывать к определенному IP-адресу, который связан только с этим процессом. Это можно сделать при помощи стандартного средства операционной системы Linux ifconfig, которое позволяет выделение нескольких IP-адресов одному узлу или, точнее, одному сетевому интерфейсу (так называемый IP-aliasing). Большинство современных реализаций IP-стека позволяют это делать. И когда требуется перенести процесс на другой узел, перенести его вместе с этим IP-адресом. Это тоже достаточно легко реализуется, так как использование утилиты ifconfig для снятия IP-адреса с сетевого интерфейса, равно как и для создания IP-alias на сетевом интерфейсе другого узла, не приводит ни к каким коллизиям внутри операционной системы даже во время сетевого взаимодействия через указанный IP-адрес. Что происходит с установленным TCP-соединением при указанных выше операциях снятия и создания IP-адреса, будет рассмотрено позже.

Помимо пары [IP1:port1 – IP2:port2], которая идентифицирует соединение, для поддержания TCP-соединения используется уникальный номер очереди пакетов как часть протокола TCP. Он передается с каждым пакетом и отвечает за правильную последовательность отправляемых или приходящих пакетов. Если рассматривать более подробно, то при установке TCP-соединения отправляется так называемый SYN-пакет с уникальным номером, идентифицирующим начало очереди пакетов. Все последующие пакеты соединения нумеруются по возрастанию относительно первого SYN. Основополагающей идеей TCP-протокола является то, что получение пакета с номером X означает, что все пакеты с меньшими номерами уже были получены. В целом обмен пакетами осуществляется по следующей схеме: на другую сторону соединения отправляется пакет данных с уникальным номером, в ответ на него другая сторона отправляет подтверждение, или так называемый ACK (acknowledgement), на полученный пакет, в ответ на который она также ожидает подтверждения, возможно вместе с новой порцией данных. Если же одна из сторон не получает подтверждения, пакет отправляется снова и снова через некоторые промежутки времени, пока не будет получено подтверждение либо не истечет время ожидания, которое в большинстве Linux-систем составляет около 9 минут. Этот механизм используется TCP-протоколом для предотвращения потери данных соединения [Postel, 1981].

Таким образом, для полноценного переноса соединения или, точнее, одной его стороны на другой узел необходимо, чтобы номера пакетов, отправляемых в сеть уже с нового узла, в точности соответствовали номерам пакетов ранее существовавшего соединения. Это достигается в нашей работе за счет сохранения структуры сокета в ядре Linux. Она включает в себя: структуру tcp_opt, как раз и отвечающую за все параметры соединения, включая указанные номера очередей; и буферы пакетов, уже пришедших на узел, но не обработанных приложением, а также отправленных приложением, но по каким-то причинам не отправленных в сеть. Это обеспечивает отсутствие потери данных при переносе, что является основополагающей идеей протокола TCP.

Ввиду поставленного требования переноса открытого TCP-соединения абсолютно прозрачно для удаленной стороны возникает еще одна серьезная проблема: каким образом остановить процесс, а значит, и соединение, на нашей стороне и потом восстановить его на новом узле, чтобы удаленная сторона ничего не «поняла»?

Эта проблема решается следующим образом. Перед остановкой процесса, непосредственно во время его выполнения, мы просто снимаем IP-alias с сетевого интерфейса с помощью ifconfig. После этого все пакеты, приходящие с удаленной стороны, будут уничтожены либо на нашем узле, либо на ближайшем к нам маршрутизаторе в силу отсутствия IP-адреса нашей стороны соединения в подсети. Следовательно, удаленная сторона не будет получать подтверждения на отправляемые пакеты. В результате удаленная сторона будет воспринимать ситуацию как временные проблемы с сетью и периодически отправлять повторные пакеты в течение некоторого достаточно длинного промежутка времени. Ближайший к нам маршрутизатор, не обнаружив IP-адреса в подсети, также в течение некоторого времени будет отправлять широковещательный ARP-запрос для определения расположения исчезнувшего IP-адреса.

Но тут возникает еще одна проблема: как остановить процесс, чтобы в сеть не отправились пакеты о завершении соединения, как это происходит при нормальном завершении процесса. Для этого после приостановки процесса сигналом SIGSTOP устанавливаем состояние сокета, связанного с соединением, в TIMEWAIT в ядре вручную. Это такое состояние, в котором сокет ждет последнего входящего пакета о завершении соединения и, не дождавшись его, просто завершает соединение в одностороннем порядке, не отправляя в сеть ни одного пакета. Так происходит остановка процесса.

Когда же нам необходимо восстановить соединение на новом узле, мы создаем IP-alias, связанный с процессом, уже на сетевом интерфейсе нового узла. После этого в ответ на широковещательный ARP-запрос маршрутизатора будет отдан уже новый Ethernet адрес расположения используемого нами IP-alias. Теперь повторный пакет с удаленной стороны дойдет до нашего, уже восстановленного, процесса, а затем удаленная сторона получит подтверждение получения пакета. Далее продолжится обмен пакетами в обычном режиме.

Возвращаясь к сохранению структуры сокета в ядре, отметим, что сохраняются и восстанавливаются на новом узле не все поля этой структуры, а только те, которые связаны с транспортным протоколом TCP. К ним относится структура tcp_opt и очереди TCP, связанные с данным соединением, назначение которых было описано выше, адрес и порт нашей и удаленной стороны соединения, а также некоторые флаги.

Приведем полный алгоритм переноса открытого соединения (здесь опускается алгоритм сохранения и восстановления процесса, в силу того что для этих целей был взят механизм CRAK [Zhong, Nieh, 2001], разработанный в Columbia University, в ноябре 2001):

Запуск процесса:

– перед тем как запустить процесс, на сетевом интерфейсе создается новый IP-alias, и запущенный процесс привязывается к этому IP-адресу.

Перенос процесса:

– снятие IP-адреса с узла;
– установка состояния сокета в TIMEWAIT в ядре;
– сохранение процесса в файлы;
– сохранение полей структуры сокета в ядре, в частности структуры tcp_opt, буферов TCP, относящихся к данному соединению, адресов и портов, в файлы;
– перенос файлов на другой узел;
– восстановление процесса на новом узле;
– создание нового сокета на новом узле;
– замена полей нового сокета на сохраненные поля, запись сохраненных буферов в очереди TCP;
– создание используемого IP-адреса уже на новом узле.

Следует также отметить, что многие прикладные программы могут быть привязаны к определенному IP-адресу, обычно это осуществляется посредством конфигурационного файла, например параметр bind_to, или параметрами командной строки, как у известного клиента ssh.

Преимущества и недостатки предлагаемого решения

Преимущества. Основное преимущество предлагаемого решения заключается, как уже было сказано выше, в том, что он дает возможность переносить процессы вместе с их открытыми соединениями абсолютно прозрачно для удаленной стороны.

Он позволяет не нагружать систему лишними агентами как просто для поддержания соединения во время переноса процессов, так и для маршрутизации данных соединения после переноса, а также не производить чрезмерную нагрузку на сеть во время миграции.

Кроме того, он может применяться в системах с несколькими точками доступа извне, чего не могут механизмы на основе proxy.

Недостатки. Основным на данный момент недостатком представленного алгоритма является ограниченное количество процессов в системе, способных мигрировать, ввиду ограниченного количества IP-alias, которые можно создать на одном Ethernet (на сегодняшний день в большинстве Linux-систем это 256), а также за счет ограниченности количества IP-адресов, выделяемых на подсеть.

К счастью, эта проблема в будущем может быть решена посредством уже разрабатываемого протокола IPV6, позволяющего значительно расширить объем IP-адресов в Internet. А количество IP-alias – это параметр, зависящий только от операционной системы, и впоследствии при необходимости может быть изменен в сторону увеличения.

Данный алгоритм не может применяться для переноса процессов между подсетями, так как его основной идеей является то, что ближайший к нам маршрутизатор сам разрешает проблему с динамическим обновлением физического расположения IP-адреса. Если же IP-адрес исчезнет в одной подсети и появится в другой, динамическое определение будет невозможным.

Однако большинство систем, использующих балансировку загрузки, довольно редко используют несколько подсетей, так как перенос процессов между подсетями – достаточно «дорогостоящая» операция. К тому же в развивающемся протоколе IPV6 предусмотрено стандартное средство для решения этой проблемы, поэтому в будущем, возможно, данное ограничение будет снято.

Еще одним недостатком является ограниченное время на перенос процесса на другой узел в силу ограниченного времени отправки повторных пакетов с удаленной стороны, или так называемого timeout, а также ограниченного времени отправки ARP-запросов маршрутизатора, по истечении которого удаленной стороне сообщается, что данного IP-адреса не существует.

В большинстве случаев это не является проблемой, так как в современных Linux-системах timeout отправки повторных пакетов составляет около 9 минут, а timeout на ARP-запросы еще больше. В условиях хорошей сети это позволит перенести достаточно объемные процессы. Хотя в случае сильной загруженности сети между узлами это может привести к невозможности переноса. И если учесть то, что перенос процессов в основном используется для балансировки загрузки кластерных систем и систем серверов, находящихся достаточно близко друг к другу, то влияние этой проблемы на применимость алгоритма можно считать достаточно малым.

Среди недостатков алгоритма также можно отметить тот факт, что запускающий процесс человек должен заранее знать IP-адрес, к которому процесс будет привязан. Это налагает некоторые ограничения на удобство использования систем, применяющих перенос процессов, однако, не является большим препятствием к применению данного алгоритма, потому как в будущем планируется его модификация для автоматизации задания IP-адреса процесса.

И последним недостатком является требование некоторых правил программирования, таких как использование явной привязки к IP-адресу вместо использования привязки по умолчанию.

По сути это не является проблемой, так как многие современные сетевые приложения уже умеют это делать.

Реализация

Разработанный алгоритм был реализован в виде двух частей: модуля ядра Linux и приложения Linux на пользовательском уровне.

Модуль ядра выполнен на языке программирования C, приложение на пользовательском уровне – на языке C++.

В качестве переносимых процессов использовались написанные на языке C простейшие приложения клиент-сервер.

Структурная декомпозиция программной реализации

Модуль ядра предназначен для сохранения и восстановления процессов, включая сохранение и восстановление состояния сокетов. Он является модификацией модуля ядра CRAK (модифицированы механизм сохранения / восстановления сокетов и встроенная поддержка переноса открытых соединений).

Приложение на пользовательском уровне необходимо для решения широкого круга задач, связанных с переносом процесса, в частности для сохранения / восстановления IP-alias, инициирования сохранения / восстановления процесса с помощью вышеописанного модуля, переноса файлов процесса на другой узел, обработки прерываний и неисправностей сети во время переноса.

Описание работы программной реализации

Для того чтобы понять, каким образом посредством двух описанных частей реализуется представленный выше алгоритм, рассмотрим следующую схему (рис. 2).

Инициирующий процесс переноса механизм (это может быть либо человек, либо система, запускающая перенос) запускает менеджер переноса с обязательным параметром – client или server, что означает: требуется перенести процесс или запустить сервер для приема входящих

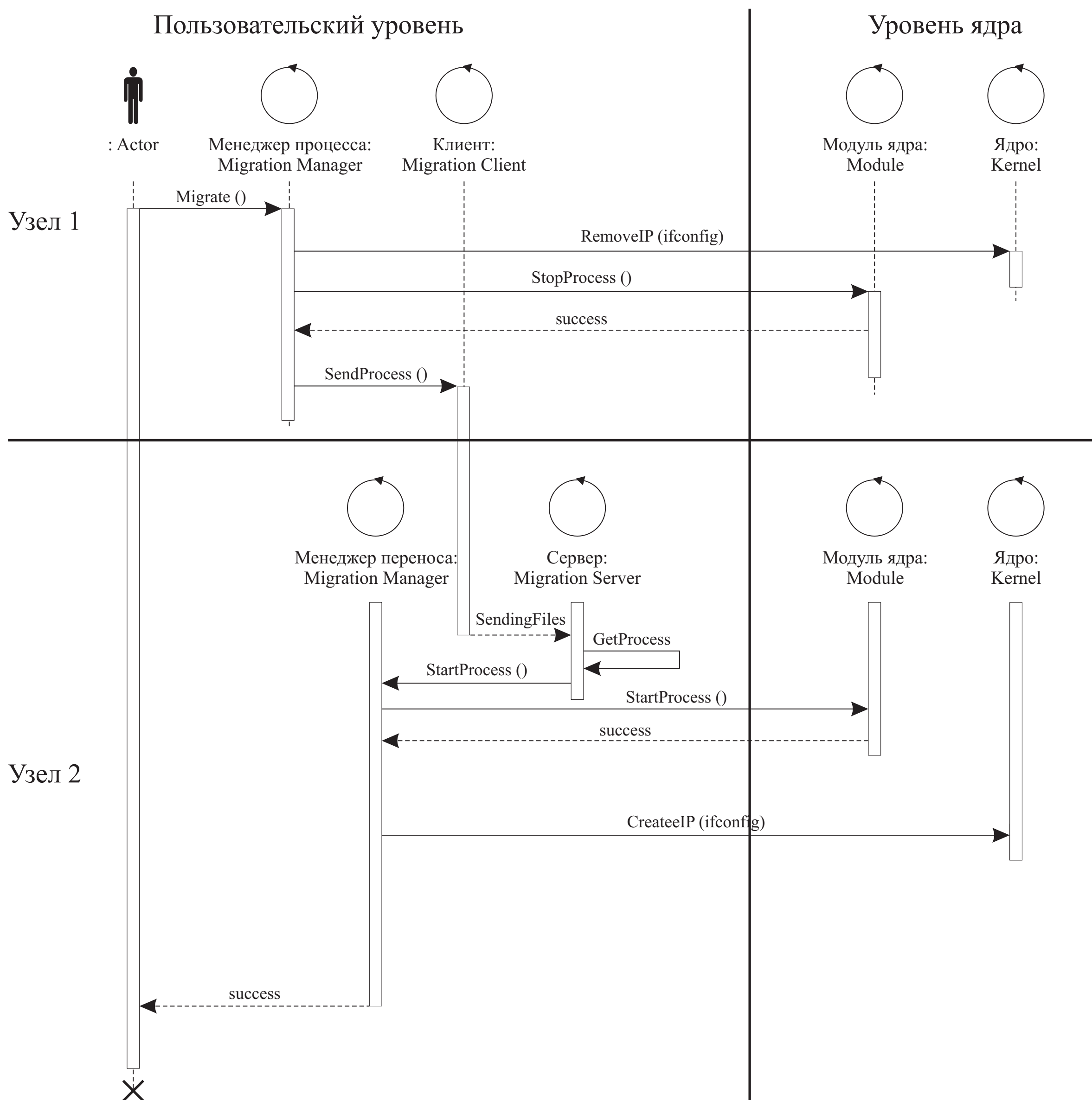

Рис. 2. Схема работы алгоритма переноса

процессов. Рассмотрен случай, когда на втором узле сервер уже запущен, а запускающему требуется перенести процесс (см. рис. 2).

После запуска менеджер переноса удаляет IP-адрес с узла посредством команды ifconfig, которая является запросом к ядру (RemoveIP).

Завершив удаление, он отправляет команду подгруженному модулю ядра для сохранения нужного процесса (метод StopProcess класса MigrationManager, на рисунке представлен как StopProcess). Описание механизма сохранения процесса посредством модуля ядра было представлено ранее.

Получив положительный результат от модуля, идентифицирующий удачное завершение сохранения, отправляет запрос клиенту (метод SendProcess класса MigrationClient, на рисунке представлен как SendProcess) на отправку файлов процесса с узла 1 на узел 2.

Уже запущенный на узле 2 сервер принимает файлы процесса (метод GetProcess класса MigrationServer, на рисунке GetProcess).

Получив все файлы, сервер информирует менеджера переноса на узле 2 о необходимости восстановить процесс (метод StartProcess класса MigrationManager), передав ему расположение файлов переносимого процесса.

После чего менеджер переноса отправляет команду модулю ядра уже на узле 2 о восстановлении процесса.

И, получив положительный ответ от модуля, означающий, что процесс удачно восстановлен, менеджер восстанавливает IP-адрес на узле 2 вызовом ifconfig (на рисунке CreateIP). Процесс восстановлен.

Возможные усовершенствования

В перспективе планируется:

– улучшение алгоритма, а конкретно решение проблемы со временем переноса, которое в настоящий момент очень ограничено, путем добавления в него отправки на удаленный узел пакета с установленным в ноль окном передачи (флаг rcv_wnd структуры tcp_opt). Этот механизм, предусматриваемый протоколом TCP, позволит значительно увеличить время переноса;

– автоматизация выделения IP-адресов процессам;

– доработка программной реализации для предоставления возможности переносить несколько процессов одновременно. Это необходимо для сохранения/восстановления программных каналов при переносе.

Список литературы

Barak A, *La'adan O.*, *Shiloh A.* Scalable Cluster Computing with MOSIX for Linux. Jerusalem: The Hebrew University of Jerusalem, 1998.

Maltz D., *Bhagwat P.* MSOCKS: architecture for transport layer mobility // In Proc. IEEE Infocom. 1998. P. 1037–1045.

Osman S., *Subhraveti D.*, *Su G.*, *Nieh J.* The Design and Implementation of Zap // Proc. of the 5th Symp. on OSs Design and Implementation (OSDI 2002). Boston, MA, 2002. P. 361–376.

Postel J. Transmission Control Protocol – DARPA Internet Program Protocol Specification, RFC 793. DARPA, 1981.

Zhong H., *Nieh J.* CRAK: Linux Checkpoint/Restart As a Kernel Module. Technical Report CUCS-014-01. N. Y.: Columbia University, 2001.

Материал поступил в редколлегию 20.04.2007