САНКТ - ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Математико-механический факультет

Кафедра системного программирования

Перенос драйвера блочного устройства DST на уровень Ethernet для проекта Cirrostratus

Курсовая работа студента 361 группы

Колянова Дмитрия Андреевича

Научный руководитель,
аспирант кафедры ВТ СПбГУ ИТМО,
ст. инженер-разработчик EMC .... /Богатырев С.В./
/подпись/

Санкт-Петербург
2010

Содержание

Введение......3
Часть1. Обзор методов и технологий......5
  Использование буфера сокета Linux......5
  Использование сокетов ядра......7
Часть2. Устройство драйвера и его реализация на Ethernet......9
  Устройство модуля пользователя......9
  Устройство модуля ядра......11
    Сценарий поведения сервера......11
    Сценарий поведения клиента......12
    Интерфейсы сокетов......12
    Адресация......13
    Транзакции......14
Результаты......15
Список использованной литературы......16

Введение

Во многих организациях главным активом является информация, поэтому нужно обеспечивать надежное хранение информационных ресурсов и быстрый доступ к ним.

В настоящее время повышенный интерес вызывает распределенное хранение данных, обладающее рядом преимуществ перед традиционным прямым подключением дисковых массивов к серверам: сеть хранения данных представляет собой архитектурное решение для подключения внешних устройств хранения данных, таких как дисковые массивы таким образом, чтобы операционная система распознала подключённые ресурсы как локальные.

Основные преимущества:

- Высокая масштабируемость
- Высокая производительность и надежность
- Простота администрирования
- Эффективное восстановление работоспособности после сбоя

На данный момент существует множество решений, но на небольших предприятиях они являются редкостью из-за слишком высокой стоимости. Целью проекта является создание надежной сети хранения данных, работающей на дешевом, доступном оборудовании и обладающей всеми вышеуказанными свойствами.

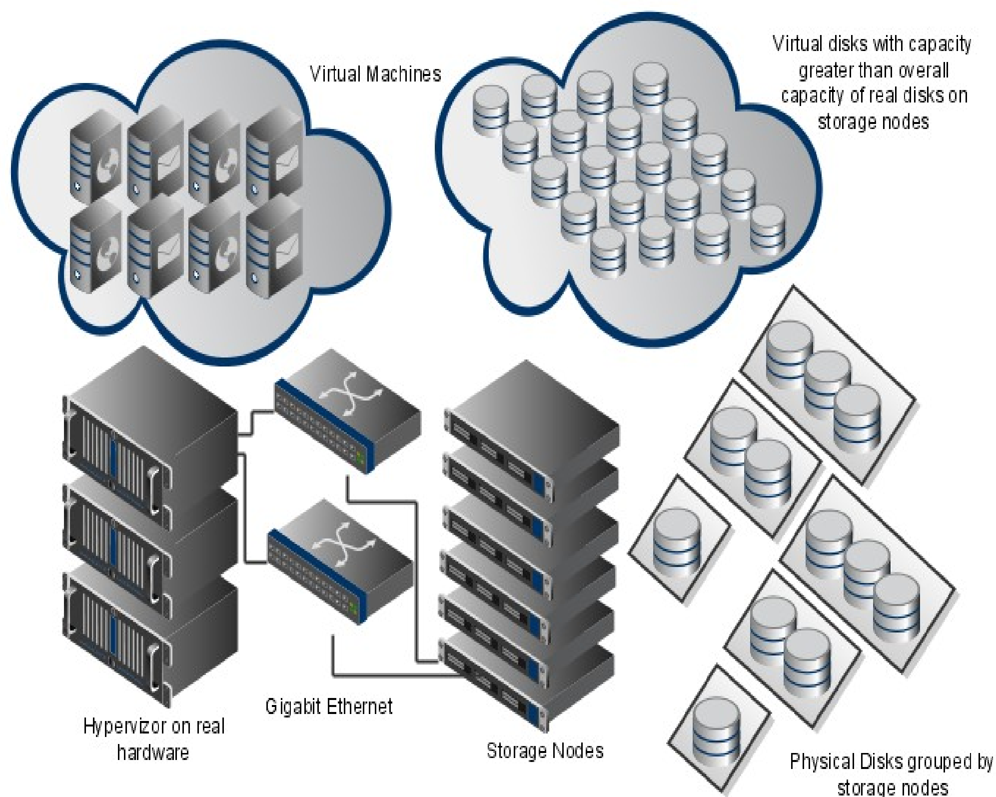

Рис.1 Распределенное хранилище

В сети, проиллюстрированной выше есть несколько серверов, на которых запущены виртуальные машины. Виртуальная машина выделяется по запросу пользователя в качестве вычислительного ресурса и каждой машине соответствует свой виртуальный диск.

Сервера с машинами соединены с хранилищем данных при помощи технологии Gigabit Ethernet. Хранилище данных, в свою очередь состоит из узлов хранения. На каждом узле установлена своя операционная система, и есть набор физических дисков.

Виртуальная машина обычно не посылает команды устройства, а запрашивает с диска блоки данных длиной, кратной странице памяти (обычно 4Кб), а дальнейшие операции с данными производятся уже на уровне страничного кеша гостевой ОС. Так как с серверной стороны мы имеем дело не с физическим диском, а с ПО, обрабатывающим наши запросы, то нет необходимости тратить ресурсы на формирование и разбор команд (например таких, как ATA или FC), что позволит поднять пропускную способность и уменьшить время отклика.

Доступ к дискам виртуальных машин по сети обычно осуществляется с использованием стека протоколов TCP/IP, что также негативно сказывается на производительности. При этом в большинстве случаев и сервер виртуальных машин и сервер хранения находятся в одном Ethernet сегменте. Использование протокола обмена данных напрямую через Ethernet-кадры позволило бы значительно увеличить производительность дисковой подсистемы виртуальных машин.

Вследствие вышесказанного, для реализации такой сети за основу взят драйвер DST, написанный для операционной системы Linux (на языке си) [4] . DST — драйвер сетевого устройства транспортного уровня для организации распределенных хранилищ данных, функционирующий в ядре на уровне блочного устройства [7]. Блочное устройство (block device) — вид файла устройств в UNIX/Linux-системах, обеспечивающий интерфейс к устройству, реальному или воображаемому, в виде файла в файловой системе [6]. То есть DST умеет общаться с блочными устройствами через сеть, и работает при помощи протоколов TCP/IP. Задачей курсовой работы является перенос DST на уровень Ethernet с целью подготовки к реализации разработанного в проекте протокола [4] и увеличения производительности системы.

Часть1. Обзор методов и технологий

Портирование драйвера велось при помощи двух виртуальных машин, на каждой из которых была установлена операционная система Linux Ubuntu, версия ядра — 2.6.31. Для запуска виртуальных машин была использована программа Virtual Box, с помощью которой можно задавать адаптеры, тип виртуальных сетевых карт и мак адреса для машин.

Машины были связаны между собой внутренней сетью, а также были связаны с интернетом. В качестве системы контроля версий использовался Git, а сам проект располагается на сайте Github [5]. Для отладки был использован Wireshark - анализатор сетевого трафика для сети Ethernet, а также встроенные утилиты операционной системы Linux.

Для организации обмена на уровне Ethernet в Linux существует два известных подхода, описанных далее.

1. Использование буфера сокета Linux

Перемещение данных для сокетов происходит при помощи основной структуры под названием буфер сокета (sk_buff). В sk_buff содержатся данные пакета и данные о состоянии, которые охватывают несколько уровней стека протокола. Каждый отправленный или полученный пакет представлен в sk_buff.

В версии ядра 2.6.31 основные поля структуры sk_buff выглядят так:

```
Struct sk_buff{
	struct sk_buff      *next;
	struct sk_buff      *prev;
	struct sock         *sk;
	struct net_device   *dev;
	__be16              protocol;
	unsigned char       *data;
	. . . . . . . . . . . . . .
};
```

Как можно заметить, несколько структур sk_buff для данного соединения могут быть связаны вместе при помощи полей next и prev – указателей на предыдущий и следующий элемент в списке или очереди пакетов. Поле dev является указателем на устройство, принявшее пакет, или устройство, через которое он будет передан. В поле data

размещается содержимое пакета. При выбранном способе реализации заголовки соответствующих уровней нужно собирать вручную.

Основные функции для работы с буферами сокетов:

```
struct sk_buff *dev_alloc_skb(unsigned int length);
```

Служит для выделения памяти под структуру `sk_buff` а также инициализации требуемых переменных.

```
unsigned char* skb_put(struct sk_buff *skb, int len);
```

Функция возвращает указатель на начало нового блока данных.

```
void skb_queue_tail(struct sk_buff_head *list_, struct sk_buff *newsk)
```

Функция помещает буфер в конец указанной очереди.

```
struct sk_buff *skb_dequeue(struct sk_buff_head *list_)
```

Извлекает первый буфер из очереди.

```
int  dev_queue_xmit(struct sk_buff);
```

Функция, отвечающая за передачу пакета.

Используя такой минимальный набор функций можно создать пакет, добавить его в очередь, заполнить заголовки и содержимое, а затем отправить по сети.

Для приема можно использовать функцию:

```
void dev_add_pack(struct packet_type *pt);
```

которая будет ждать пакет на указанном интерфейсе, и, в случае его прибытия вызовет указанную принимающую функцию. Ссылки на интерфейс и принимающую функцию содержатся в структуре `packet_type.`

Посредством прямого обращения к буферам сокетов работает один из наиболее близких по характеристикам драйвер AoE. AoE передает данные на уровне Ethernet, но не взаимодействует напрямую с блочным устройством, поэтому пакеты в нем перегружены заголовком ATA команд. Были написаны соответствующие модули, и изначально предлагалась идея сделать такой же обмен данными в драйвере DST, но, поскольку в DST использовался более высокоуровневый механизм — сокеты ядра, значительно легче было бы использовать подход, описанный далее.

Еще один аргумент против - большая вероятность того, что в одной из следующих версий ядра изменится код структуры `sk_buff.` Это означает, что код будет сложно поддерживать. Поскольку время программиста дороже времени машины, было принято решение отказаться от такого подхода.

2. *Использование сокетов ядра*

Сокет — название программного интерфейса для обеспечения обмена данными между процессами. Процессы при таком обмене могут исполняться как на одной машине, так и на различных машинах, связанных между собой сетью. Сокетный интерфейс позволяет избавиться от низкоуровневых системных вызовов. В линуксе с сокетом связаны три параметра — домен, тип и протокол. Домен определяет пространство адресов, в котором располагается сокет и множество протоколов, которые используются для передачи данных. Тип сокета определяет способ передачи данных по сети. Параметр протокола определяет протокол, используемый для передачи данных.

В качестве примера доменов рассмотрим следующие параметры:

`AF_UNIX` – используется для обмена данными между процессами.

`AF_INET` – соответствует интернет-домену.

`AF_PACKET` — используется для приема и передачи необработанных пакетов на канальном уровне.

Среди типов сокетов нас интересуют:

`SOCK_DGRAM` – передача данных в виде отдельных сообщений(датаграмм).

`SOCK_RAW` — так называемые «сырые» сокеты, требуется собирать заголовки пакетов вручную.

Сокеты можно делить соответственно на сокеты уровня ядра и сокеты уровня пользователя. Нас интересуют сокеты уровня ядра. В зависимости от параметров, заданных при создании, у сокета может быть разный функциональный интерфейс. Рассмотрим основные функции для сокетов в ядре:

```
int sock_create (int family, int type, int protocol, struct socket
**res)
```

Функция создает сокет с заданными параметрами.

```
int kernel_recvmsg(struct socket *sock, struct msghdr *msg, struct
kvec *vec, size_t num, size_t size, int flags);
int kernel_sendmsg(struct socket *sock, struct msghdr *msg, struct
kvec *vec, size_t num, size_t size);
```

Функции, отвечающие за прием/передачу пакетов.

```
int kernel_bind(struct socket *sock, struct sockaddr *addr, int
addrlen);
```

Эта функция используется для явного связывания сокета с некоторым адресом, в

зависимости от домена сокета, в качестве адреса может выступать ip – адрес, мак – адрес, или просто строка, обозначающая имя файла.

Для приложений, связанных с интернетом применяют сокеты третьего уровня, они предназначены для работы со стеком протоколов TCP/IP, на их основе изначально и был написан драйвер DST. Для того, чтобы передавать данные напрямую по Ethernet, нужны так называемые сокеты канального (второго) уровня. Такие сокеты называются пакетными , для их создания необходимо задать следующие параметры:

```
sock_create(AF_PACKET, SOCK_RAW, htons(ETH_P_ALL));
```

тип сокета указывает на то, что при приеме пакетов заголовки будут переданы программе без каких-либо изменений, что и требуется для реализации собственного протокола. Для решаемой задачи недопустимо использовать пакеты `SOCK_DGRAM`, поскольку, не имея функции `connect` и не имея заголовков, невозможно определить исходный адрес отправителя [8]. Установлено, что при использовании пакетных сокетов можно получить выигрыш в производительности [3].

Сокетный интерфейс стабильный и устоявшийся, его легко поддерживать и читать, сокеты мало проигрывают относительно `sk_buff`. Так как альтернатив больше нет, был выбран такой подход.

Часть2. Устройство драйвера и его реализация на Ethernet

Драйвер DST может выполнять роль клиента и сервера. В его состав входят две программы — модуль ядра и модуль пользователя. Для запуска драйвера на машине, представляющей сервер нужно добавить виртуальный диск, подключить модуль ядра, затем сконфигурировать модуль ядра из пространства пользователя, тем самым экспортировав диск и указав файл с правами доступа для клиентов. На клиенте соответственно делается то же самое — подключается и конфигурируется модуль ядра. Когда все запущено, с клиента можно создать файловую систему на только что подключенном виртуальном диске, после чего он будет отображаться на клиенте — дальше с диском можно работать в обычном режиме.

1. *Устройство модуля пользователя*

Модуль пользователя (в дальнейшем `dst-userspace`) представляет собой средство с интерфейсом командной строки. При помощи `dst-userspace` можно сконфигурировать модуль ядра, при этом разрешается использовать следующие команды:

- Экспорт диска, с указанием его имени и ip-адреса и порта к которому этот диск будет привязан, а также указывается файл в котором содержатся адреса доверенных клиентов.
- Соединение с удаленным диском, с указанием адреса и порта.

Также в командах можно указывать ряд других опций, в том числе опции для шифрования данных. Полученные от пользователя порт и адрес помещаются в адресную структуру `sockaddr_in`:

```
struct sockaddr_in {
    short int          sin_family;  // Семейство адресов
    unsigned short int sin_port;    // Номер порта
    struct in_addr     sin_addr;    // IP-адрес
    unsigned char      sin_zero[8];
};
```

то есть в этой структуре содержится домен, порт и адрес. Для указания оставшихся параметров используется структура `dst_network_ctl`:

```
struct dst_network_ctl{
	unsigned int type;
	unsigned int proto;
	struct saddr addr;
}
```

в ней содержится тип сокета, протокол, и адресная структура, в которую вкладывается sockaddr_in.

Затем создается сокет между программами ядра и пользователя:

```
s = socket(PF_NETLINK, SOCK_DGRAM, NETLINK_CONNECTOR);
```

через него в ядро передается содержимое структуры dst_network_ctl, а также команды, которые указывают какое действие запросил пользователь — удаление экспортированного диска, экспортирование диска, или запрос на соединение с удаленным диском. В оригинальной версии драйвера для сокетов были заданы следующие параметры: тип — SOCK_STREAM, домен — AF_INET, протокол — IPPROTO_TCP. Эти параметры были заменены следующим образом: тип — SOCK_RAW, домен - AF_PACKET, протокол — htons(ETH_P_ALL). В связи с изменением типа сокета пришлось поменять адресную структуру, и использовать sockaddr_ll вместо sockaddr_in:

```
struct sockaddr_ll{
	unsigned short	sll_family;
	__be16		sll_protocol;
	int		sll_ifindex;
	unsigned short	sll_hatype;
	unsigned char	sll_pkttype;
	unsigned char	sll_halen;
	unsigned char	sll_addr[8];
}
```

соответственно домен и протокол записываются в поля sll_family и sll_protocol, в поле sll_addr[8] записывается мак-адрес интерфейса, в поле sll_ifindex записывается индекс интерфейса. Также были заменены все аргументы у функций, принимающих адрес и порт на мак адрес. На этом завершается изменение программы пользователя — в итоге изменились аргументы командной строки — вместо пары порт адрес аргументом принимается мак адрес интерфейса.

2. Устройство модуля ядра

Модуль ядра начинает свою работу с того, что запускает парсер конфигурации, и в зависимости от результатов ведет себя либо как клиент, либо как сервер.

2.1 Сценарий поведения сервера

В случае, если команда была на экспортирование, драйвер находит блочное устройство — диск, который указал пользователь, а затем открывает его на чтение и запись. После этого создается структура под названием `dst_state`, отвечающая за состояние, связанное с данным узлом(диском). Эта структура является одной из основных в драйвере, поэтому стоит рассмотреть ее подробнее: она содержит указатель на структуру, хранящую информацию о диске к которому привязано данное состояние, адресную структуру, в которой, как уже говорилось ранее содержится порт и ip-адрес, уровень доступа для данного клиента, а также сокет через который происходит обмен данными между машинами.

```
struct dst_state
{
     struct dst_node          *node;
     /* Network address for this state */
     struct dst_network_ctl   ctl;
     /* Permissions to work with: read-only or rw connection */
     u32            permissions;
     struct socket       *socket;
     /* Cached preallocated data */
     void           *data;
     /* Currently processed command */
     struct dst_cmd      cmd;
     . . . . . . . . . . . . . . .
};
```

После выделения структуры в ней инициализируется слушающий сокет, который ждет запросов на конфигурацию от клиентов. Запускается поток, в котором должны обрабатываться все такие запросы. В потоке при получении запроса проверяется уровень доступа для данного клиента, и, в случае, если доступ разрешен, создается новое состояние, представленное также структурой `dst_state`, отправляется пакет с ответом на команду конфигурации, размером и именем диска, затем запускается поток, который отвечает за

соединение с данным клиентом. То есть в результате есть слушающее состояние, которое принимает новых клиентов, а все остальные состояния отвечают за обмен данными между конкретным клиентом и сервером.

2.2 Сценарий поведения клиента

В случае, если от пользователя была получена команда на соединение, клиент выделяет структуру `dst_state` для подключаемого диска, в нем инициализируется сокет, сокет коннектится к слушающему сокету сервера, затем отправляется запрос на конфигурацию, принимается ответ, и, если есть доступ к данному диску, то на клиенте создается виртуальный диск с такого же размера как и на сервере. После всего этого запускается поток в котором обрабатывается обмен данными между клиентом и сервером.

2.3 Интерфейсы сокетов

Трудности в переносе драйвера на уровень ниже были вызваны значительными отличиями в интерфейсе сокетов 2-го и 3-го уровней, рассмотрим функции, которые были использованы в драйвере:

```
int kernel_bind(struct socket *sock, struct sockaddr *addr, int addrlen);
```

Данная функция привязывает сокет к интерфейсу, заданному в структуре `sockaddr`, то есть указывает с какого сетевого интерфеса сокет должен принимать пакеты. Данная функция поддерживается в обоих типах сокетов и, поэтому не требует никаких изменений.

```
int kernel_connect(struct socket *sock, struct sockaddr *addr, int addrlen, int flags);
```

Функция используется для установки соединения, автоматически привязывая сокет к заданному интерфейсу. В случае сокетов второго уровня, такая функция во-первых не поддерживается, а во-вторых в принципе не нужна, поэтому в местах, где она встречается она либо удалена, либо заменена на функцию `kernel_bind`.

```
int kernel_accept(struct socket *sock, struct socket **newsock, int flags);
```

Функция создает для общения с клиентом новый сокет, с теми же параметрами, от которого он клонирован, эта функция работает только для сокетов типа TCP/IP. Соответственно для сокетов второго уровня можно заменить ее на пару функций `sock_create` и `kernel_bind`.

```
int kernel_listen(struct socket *sock, int backlog);
```
Данная функция переводит сокет в слушающее состояние. Для сокетов второго уровня неприменима, но и нет необходимости ее использовать, поэтому эту функцию можно просто убрать из рассмотрения.
```
int kernel_recvmsg(struct socket *sock, struct msghdr *msg, struct kvec *vec, size_t num, size_t size, int flags);
int kernel_sendmsg(struct socket *sock, struct msghdr *msg, struct kvec *vec, size_t num, size_t size);
```
Функции, отвечающие за прием/передачу пакетов, работают одинаково для обоих типов сокетов, поэтому не требуют внесения изменений.
```
int kernel_sendpage(struct socket *sock, struct page *page, int offset, size_t size, int flag);
```
Функция посылает страницы памяти по сети, для сокетов второго уровня реализация этой функции неприменима, поэтому пришлось переписать ее, используя функцию `kernel_sendmsg`, и разбивать страницу на пакеты.
```
int kernel_getpeername(struct socket *sock, struct sockaddr *addr, int *addrlen);
```
Функция смотрит на то что пришло на сокет и записывает адрес отправителя в поле `addr`. Не поддерживается для сокетов второго уровня, альтернатива — принять пакет при помощи функции `kernel_recvmsg` и, обработав пакет, посмотреть на адрес отправителя.

2.4 Адресация

Как и в пространстве пользователя, в ядре также используется структура `sockaddr_ll` вместо `sockaddr_in`. Поскольку сокеты второго уровня используют для обмена только необработанные пакеты, приходится держать в начале буфера для каждого состояния Ethernet – заголовок, и копировать в буфер данные. Поэтому в структуру `dst_state` добавлены поля, которые хранят мак-адреса отправителя и получателя. Для слушающего состояния добавлен список мак-адресов принятых клиентов. В случае сервера мак адрес клиента определяется по пришедшему пакету, свой мак адрес указывается пользователем. В случае клиента свой мак адрес получается по указанному интерфейсу, адрес сервера указывается пользователем.

Соответственно приходится разбирать заголовки вручную, нет автоматической возможности узнать откуда пришел пакет. Поскольку отсутствует реализация для функции

`kernel_connect`, получается, что все пакеты со всех клиентов приходят на все сокеты. То есть каждый раз при поступлении пакета на сокет, выполняется сравнение мак адреса отправителя с мак адресом или списком мак адресов состояния соответственно. Если мак-адрес не встретился в списке, то запускается новый поток для нового клиента. Такой подход может плохо сказываться на производительности, но на текущий момент нас это устраивает.

2.5 Транзакции

Обмен данными организован в виде транзакций – неделимой отправки нескольких страниц памяти в сеть. На каждую транзакцию должно приходить подтверждение. В первоначальной версии драйвера принцип был следующим: транзакции посылались друг за другом, и, так как раньше соединение осуществлялось с помощью протокола TCP, подтверждение могло не произойти только в случае разрыва соединения, потому что протокол гарантировал доставку пакетов. В случае разрыва отправлялась команда на пинг, устанавливалось соединение, а затем, в случае истечения определенного интервала времени (таймаута), восстанавливалась незавершенная транзакция. В текущей реализации операции повторного запроса на соединение не поддерживаются, так как в них нет необходимости.

Результаты

На данный момент достигнута минимальная поставленная задача - драйвер работает на уровне Ethernet.

Существуют следующие проблемы:

- Никакой гарантии доставки Ethernet пакетов нет, пакеты могут теряться. В связи с этим транзакции часто не завершаются - приходится ждать истечения таймаута, а затем восстанавливать транзакции — это отнимает много времени, потому что потеря пакетов происходит достаточно часто.
- Любой пакет, который приходит на интерфейс, копируется по числу открытых сокетов — выполняется много лишних проверок на соответствие мак адресов. При росте числа клиентов растут затраты на такие проверки.

По причинам, указанным выше, производительность снизилась в несколько раз. Эти проблемы должны быть решены внедрением протокола, разработанного в проекте [4], поскольку протокол включает в себя механизм для гарантии доставки пакетов. Драйвер сохранил возможность обмена данными с блочными устройствами по сети, но тем не менее требуется тщательное тестирование на нескольких клиентах.

Текущие направления работы:

- Повышение производительности драйвера.
- Реализация протокола.

Дальнейшие возможные направления работы:

- Реализация карты сети.
- Реализация кэшей.
- Разработка и реализация алгоритма для распределения данных.
- Реализация Heatmap.

Список использованной литературы

[1] Sage A . Weil, Scott A. Brandt, Ethan L. Miller, Carlos Maltzahn. CRUSH: Controlled, Scalable, Decentralized Placement of Replicated Data.

[2] Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman. Linux device drivers, Third Edition.

[3] Кузнецов К. О. Анализ, сравнение и адаптация протоколов для оптимальной передачи данных в проекте Cirrostratus.

[4] Лапин С. К. Разработка надежного протокола обмена данными на уровне Ethernet в проекте Cirrostratus.

[5] Сайт веб-сервиса Github // https://github.com/

[6] Сайт, посвященный UNIX/Linux системам // http://xgu.ru/

[7] Сайт с описанием DST // http://www.ioremap.net/projects/dst

[8] Сайт энциклопедии сетевых протоколов // http://www.protocols.ru/modules.php?name=News&file=article&sid=66