Санкт-Петербургский Государственный Университет
Математико-Механический факультет
кафедра системного программирования

Организация надежных соединений через виртуальные каналы

курсовая работа студента 445 группы Федора Бурдун
научный руководитель
аспирант кафедры системного програмирования Антон Бондарев

2012

Введение

В связи с необходимостью внедрения в ОСРВ Embox стека предсказуемых сетевых сообщений, было приянто решение реализации прототипа, который бы соответствовал поставленым критериям таким как (четче. Раскрыть. Может вставить список) гибкость и возможность быстро и эффективно реализовывать и внедрять новые протоколы. Но все же основной задачей данной подсистемы является реализации возможности гарантировано быстро обрабатывать критически важные сетевые пакеты в так называемых узких локальных сетях. Подобная задача возникает в рамках проекта управления роботами через протокол bluetooth на основе платформы Lego Mindstorm, которые активно используется кафедрой системного программирования и теории управления.

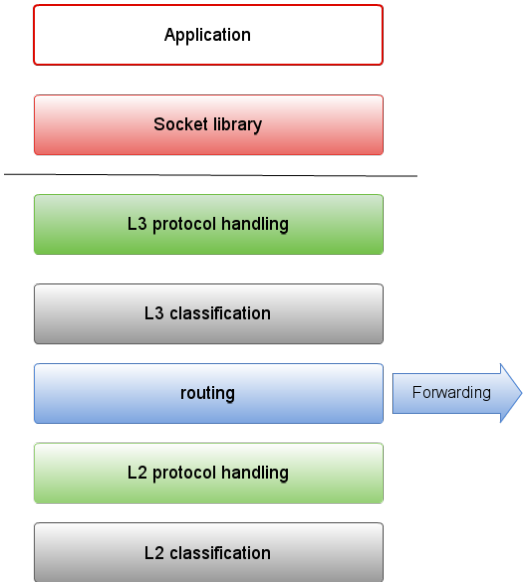
Существует два принципа доставки данных по сети: первый основан на коммутации каналов, а второй на коммутации пакетов.

При доставке данных по первому принципу установка соединения происходит единожды - строится весь путь следования данных. Это требует значительных затрат по времени, но дальнейшая доставка данных происходит по строго определенным правилам, и следовательно детерминирована по времени. Данный принцип традиционно применяется в сетях где требуется детерминированное время (телефония удаленное управление и так далее).

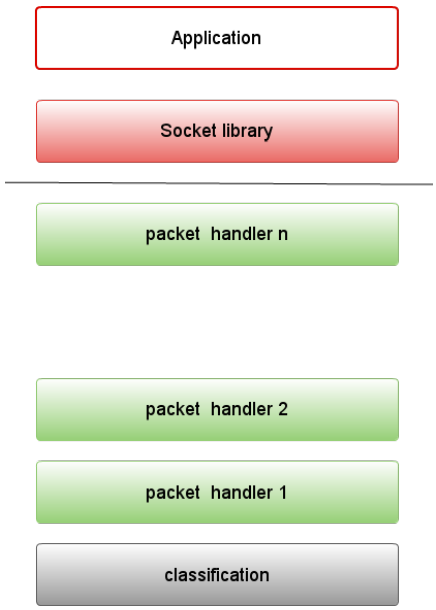
При доставке по принципу коммутации пакетов вычислять весь путь не нужно. Анализ и обработка каждого пакета происходит на каждом сетевом узле через который пакет проходит. В следствии этого анализа происходит принятие решения о том куда же направить пакет дальше и как его в дальнейшем обрабатывать. Традиционно данный принцип применяется в IP сетях и сетях с переменной средой передачи данных. Данный вид доставки данных является более масштабируемым более гибким и дешевым, но при таком подходе трудно добиться детерминированного времени доставки данных.

Основной идеей для создания сетевого стека pnet является попытка объединить преимущества двух этих типов передачи данных и добиться детерминированного времени при доставке и обработке пакета в сетях где традиционно применяется принцип пакетной коммутации.

На рисунке показан путь обработки пакетов в обычном стеке протоколов



Как видно анализ разных частей пакета происходит каждый на своем уровне и в зависимости от анализа выполняются те либо иные действия с данными. В отличии от данного подхода в pnet происходит попытка заранее выяснить все вызовы необходимые для обработки пакета и уже затем обработать пакет по полученным требованиям с заданным приоритетом



Таким образом данный подход позволяет обрабатывать заданные типы пакетов по детерменированным правилам. Данные правила могут как задаваться статически так и вырабатываться динамически во время работы системы.

Существующие решения

Поставленым критерям (гибкость конфигурирования, прозрачность разработки и т.д.) уверено отвечает сетевая подсистема Berkley Netgraph, которая успешно применяется в операционных системах семейства *BSD.

Netgraph это сетевая подсистема в ядре, следующая принципу UNIX достижения мощности посредством комбинации простых инструментов, каждый их которых предназначен для выполнения одной, вполне определенной задачи. Основная идея проста: есть узлы (nodes) (инструменты) и ребра (edges) которые соединяют пару узлов (отсюда и "граф" в "netgraph"). Пакеты данных идут в двух направлениях вдоль ребер от узла к узлу. Когда узел получает пакет данных, он обрабатывает его, и затем (обычно) отправляет его другому узлу. Обработка может быть простейшим добавлением/удалением заголовков или может быть более сложной или включает другие компоненты системы. Netgraph напоминает потоки (Streams) в System V, но он разработан более гибким и производительным.

Netgraph оказался очень полезным для работы в сети, и сейчас он используется в Whistle InterJet для различных комбинаций протоколов:

- IP пакеты, передаваемые поверх "Cisco HDLC" (по существу, пакеты дополнены двухбайтным полем Ethertype, и периодически посылаются keep-alive пакеты).
- IP пакеты, передаваемые поверх frame relay (frame relay предоставляет до 1000 виртуальных каналов точка-точка поверх одой кабельной сети).
- IP в инкапсуляции RFC 1490 поверх frame relay (RFC 1490 определяет способ передачи нескольких протоколов через одно соединение и он часто используется совместно frame relay).

- PPP поверх HDLC
- PPP поверх frame relay
- PPP в инкапсуляции RFC 1490, поверх frame relay
- PPP поверх ISDN , плюс обычный PPP поверх асинхронных последовательных (таких как модемы и терминальные адаптеры) и PPTP, которые включают шифрование.

Во всех этих протоколах данные полностью обрабатываются в ядре. В случае PPP, пакеты согласования (negotiation) обрабатываются отдельно в пользовательском режиме.

Идея комбинации различных протоколов подсказывает нам, что должны быть узлы и ребра. Менее очевиден факт, что узел может иметь определенное число подключений к другим узлам. Например, вполне возможно иметь одновременно IP, IPX, и PPP в инкапсуляции RFC 1490; конечно, мультиплексирование это та задача, для которой и нужен RFC 1490. В этом случае нужно три ребра подключить к узлу RFC 1490, одно для каждого стека протоколов. Нет требований, чтобы данные следовали строго в определенном направлении, и нет ограничений на действия, которые узел выполняет с пакетом. Узел может быть источником/потребителем данным, например, если он связан с аппаратной частью, или он может просто добавлять/удалять заголовки, мультиплексировать и т. п.

Узлы netgraph существуют в ядре и полупостоянно. Обычно узел существует пока он подключен, к какому либо другому узлу, однако некоторые узлы постоянные, например, узлы, связанные с аппаратной частью; когда число ребер уменьшается до нуля, аппаратное устройство выключается. Поскольку узлы существуют в ядре, они не связаны с каким либо определенным процессом.

Очевидно, что должны быть узлы и ребра. Менее очевиден факт, что узел может иметь определенное число подключений к другим узлам. Например, вполне возможно иметь одновременно IP, IPX, и PPP в инкапсуляции RFC 1490; конечно, мультиплексирование это та задача, для которой и нужен RFC 1490. В этом случае нужно три ребра подключить к узлу RFC 1490, одно для каждого стека протоколов. Нет требований, чтобы данные следовали строго в определенном направлении, и нет ограничений на действия, которые узел выполняет с пакетом. Узел может быть источником/потребителем данным, например, если он связан с аппаратной частью, или он может просто добавлять/удалять заголовки, мультиплексировать и т. п.

Узлы netgraph существуют в ядре и полупостоянно. Обычно узел существует пока он подключен, к какому либо другому узлу, однако некоторые узлы постоянные, например, узлы, связанные с аппаратной частью; когда число ребер уменьшается до нуля, аппаратное устройство выключается. Поскольку узлы существуют в ядре, они не связаны с каким либо определенным процессом.

В netgraph, ребра на самом деле не существуют *сами по себе*. Вместо этого ребро это просто комбинация двух крючков (hooks), по одному от каждого узла. Крючок узла определяет как узел может быть подключен. Каждый крючок имеет уникальное, статически определенное имя, которое часто отражает его цель. Имя имеет значение только в контексте данного узла; два узла могут иметь крючки с одинаковым названием.

Основные идеи

- узлы имеют «крючки» (hooks)
  «крючки» могут иметь имена

- узлы (nodes) имеют распознавать особые названия. (это облегчает конфигурирование соидинений специальными шаблонными наименованиями)
- симметричные связи между «крючками»
- «крючок» существует только если подключен к другому
- один «крючок» можно подключить только к одному «крючку»
- узел может иметь много крючков
- «крючки» могут использоваться для служебной информации или в любых других целях узла
- управляющие сообщения направляются к конечной точке, конечная точка может быть определена или абсолютным или относительным путем
- каждый узел может указать тип сообщения

Качественная характеристика Netgraph

• Простые модули с минимальными накладными расходами (не нужно волноваться об инфраструктуре, но только о том, что мы хотим реализовать)

- реализация узлов в пользовательском пространстве
- произвольный порядок узлов
- возможность иметь одтельные утилиты конфигурирования для отдельных узлов
- возможность подключения одного узла ко многим
- сопряжение со встроенными метаданными (например приоритет пакета)
- узлу может быть назначено понятное имя

Функциональная сущность

Как netgraph реализован? Одна из главных целей netgraph это *скорость*, поэтому он полностью работает в ядре. Другое конструктивное решение в том, что netgraph полностью функциональный. То есть пакеты не ставятся нигде в очередь при перемещении от узла к узлу. Вместо этого используется прямой вызов функций. Пакеты данных это packet header mbuf'ы, в то время как мета-данные и управляющие сообщения Си-структуры расположенные в куче (используя malloc типа M_NETGRAPH).

Объектно-ориентированная сущность

Netgraph отчасти имеет объектно-ориентированную архитектуру. Каждый тип узла определен как массив указателей на методы, или функции Си, которые определяют специфическое поведение узлов данного типа. Каждый метод может быть оставлен NULL для того, чтобы оставить поведение по умолчанию.

Аналогично, есть несколько управляющих сообщений, которые понимают узлы всех типов и которые обрабатываются базовой системой (они называются общими управляющими сообщениями, generic control messages). Каждый тип узлов может дополнительно определять

свои собственные управляющие сообщения. Управляющие сообщения всегда содержат typecookie и команду, которые вместе определяют, как интерпретировать это сообщение. Каждый тип узлов должен определить свое уникальное значение typecookie если предполагается, что он будет получать свои управляющие сообщения. Общие управляющие сообщения имеют предопределенные значения typecookie.

Память

Netgraph использует подсчет ссылок для структур узлов и крючков. Каждый указатель на узел или крючок считается как одна ссылка. Если узел имеет имя, оно тоже считается ссылкой. Вся связанная с netgraph область памяти выделяется и освобождается используя malloc типа M_NETGRAPH.

Синхронизация

Выполнение кода в ядре требует внимательной синхронизации. Узлы netgraph обычно выполняются через splnet() (см. spl(9)). Для большинства типов узлов не требуется дополнительного внимания. Некоторые узлы, однако, взаимодействуют с другими частями ядра, которые выполняются с другим приоритетом. Например, последовательный порт работает через spltty() и поэтому ng_tty(8) должен это учитывать. На этот случай в netgraph есть альтернативные вызовы передачи данных, которые обрабатывают все необходимые очереди автоматически.

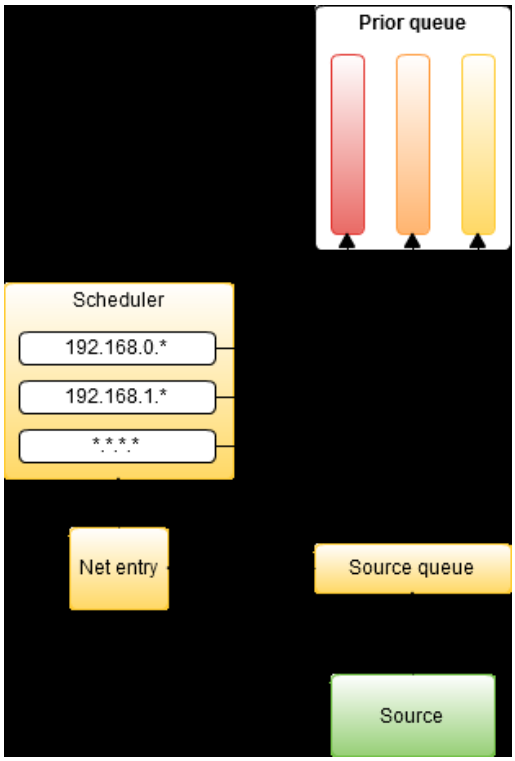
Реализация в OCPB Embox

В рамках проекта OCPB Embox реализована подсистема pnet (прототип идейно схожий с netgraph).

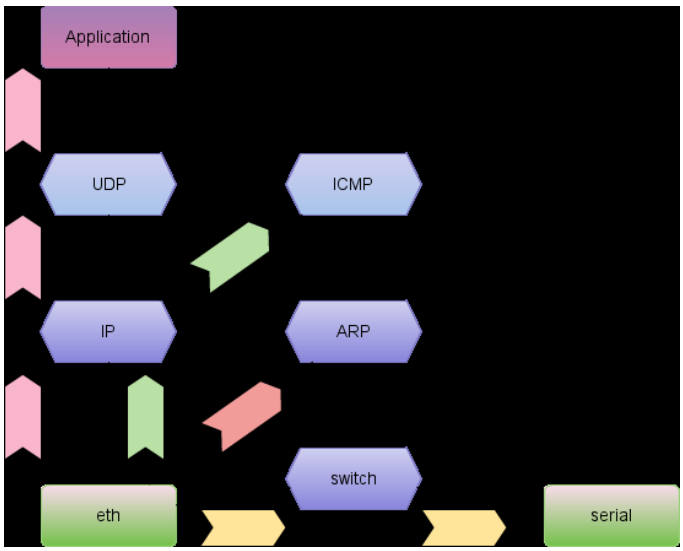
OCPB Embox для управления стеком pnet имеет консольный интерфейс, который позволяет гибко управлять подсистемой во время выполнения ОС на целевой ЭВМ.

pnet [options] [keys] graph [nodes | rules]

Так здесь могут быть добавлены различные правила для mac и ip адреса позволяющие задавать сразу целевую ноду, которой предназначаются пакеты



С учетом приоритетов в работающей систме пакеты могут распространяться следующим образом:



Создание модуля

Для создания нового модуля в подсистеме pnet необходимо реализовать и зарегистрировать обработчики следующего вида (имена методов не принципиальны и оставляются на усмотрение программиста):

```
int my_tx_hnd(struct pnet_pack *pack)
int my_rx_hnd(struct pnet_pack *pack)
```

Далее в исходном коде модуля обработчики должны быть зарегистрированы подобным образом:

```
PNET_NODE_DEF("null node", {
   .tx_hnd = my_tx_hnd,
   .rx_hnd = my_rx_hnd
});
```

В обоих вышеперечисленных методах после обработки данных должна быть вызвана функция освобождающая ресурсы:

```
pnet_pack_destroy(pack)
```

Протокол разделения каналов

В качестве протокола разделения каналов реализован протокол реализующий множественный доступ с кодовым разделением, для чего вводится понятие индентификатора канала, который инкапсулируется в заголовке соответсвующего пакета. Данная схема противопоставляется схеме с разделенным временем и имеет свои плюсы и минусы.

Плюсом данной схемы является:

- возможность если по одному из каналов не отправляются данные ресурс сети использовать другими каналами (т.е. отсутсвует простаивание ресурсов)

В свою очередь есть очевидный минус:

- возможность “наводнения” потока пакетами принадлежащими одному каналу (данная задача решается введением планировщика с элементами разделения по времени)

<table>
<tr><td colspan="3">заголовок</td><td rowspan="2">данные</td></tr>
<tr><td>тип сообщения</td><td>размер пакета</td><td>индентификатор</td></tr>
</table>

Протокол обеспечения надежности доставки

Для обеспечения надежности доставки данных реализован протокол инкапсулирующий в себе алгоритм CRC32 (cyclic redundancy code)

Алгоритм вычисления контрольной суммы, предназначенный для проверки целостности передаваемых данных. Алгоритм CRC обнаруживает все одиночные ошибки, двойные ошибки и ошибки в нечетном числе битов.

Для получения контрольной суммы, необходимо сгенерировать полином. Основное требование к полиному: его степень должна быть равна длине контрольной суммы в битах. При этом старший бит полинома обязательно должен быть равен «1».

Из файла берется первое слово. Если старший бит в слове «1», то слово сдвигается влево на один разряд с последующим выполнением операции XOR. Соответственно если старший бит в слове «0», то после сдвига операция XOR не выполняется. После сдвига (умножения) теряется старый старший бит, а младший бит освобождается (обнуляется). На место младшего бита загружается очередной бит из файла. Операция повторяется до тех пор, пока не загрузится последний бит файла.

После прохождения всего файла, в слове остается остаток, который и является контрольной суммой.

```
unsigned short Crc16(unsigned char *pcBlock, unsigned short len)
{
   unsigned short crc = 0xFFFF;
   unsigned char i;

   while (len--)
   {
      crc ^= *pcBlock++ << 8;

      for (i = 0; i < 8; i++)
         crc = crc & 0x8000 ? (crc << 1) ^ 0x1021 : crc << 1;
   }

   return crc;
}
```

заголовок			данные
тип сообщения	размер пакета	контрольная сумма	

Результаты

Таким образом в ОСРВ Embox на базе подсистемы pnet был разработан протокол переключения каналов и протокол гарантирующий надежную даставку данных, которая обеспечивается алгоритмом CRC16.

Как следствие вышесказанного применимость и полезность реализованой подсистемы pnet подтверждена. К сожалению ввиду нахождения измерительного API в стадии разработки мы не можем оперировать количественными характеристиками описывающими время работы подситсемы и сравнить ее с другими реализациями, но соответствующая оценка появится своевременно с готовностью API.

Все исходные коды проекта являются общедоступными и опубликованы под лицензией BSD.

Адрес проекта: http://code.google.com/p/embox.

Логин по которому можно оценить вклад в развитие проекта: Ignwah.

Литература

1. BSD NetGraph docs
2. Wiki проекта Embox
3. Э. Таненбаум Разработка и реализация операционных систем
4. Описание алгоритма CRC32