Indy in Depth
Глубины Indy

Indy

Taming Internet development one protocol at a time.

Оглавление

От переводчика

Зачем Я стал переводить данную книгу? Ну, это потому что по данной теме очень мало информации, особенно на русском языке. Поэтому я рискнул. Но поскольку я не профессиональный переводчик, то возможны некоторые погрешности в переводе, поэтому принимайте как есть. В конце концов, дареному коню в зубы не смотрят.

Перевод основан на старой предварительной версии книги, к сожалению, у меня нет окончательной редакции. Но даже и в этой редакции, информация приведенная в данной книге того стоит.

Об авторах, они пришли из мира Юникс, отсюда некоторая ненависть к Windows и к неблокирующим вызовам. У авторов также чувствуется некоторый хакерский и даже вирус-мейкерский подход, это видно из текста, в части приведения примера почти готового трояна, одобрение нарушения законодательства в части мер по передачи алгоритмов строгого шифрования и какими методами это было сделано. Но все это не снижает ценности данной книги. Текст, который очень простой и почти не составил сложностей с переводом, кроме некоторых мест.

В настоящее время есть три направления построения Интернет библиотек:

1. библиотеки событийно-ориентированные, большинство компонент Delphi – к этому классу относится ICS (Internet Component Suite от Франсуа Пьетте http://www.overbyte.be);
2. библиотеки с линейным кодом, структурно ориентированное программирование – к этому классу относится Indy;
3. чистые процедуры и функции, типичный представитель Synapse www.ararat.cz/synapse

Вопрос что лучше – это вопрос религиозный, мне лично нравится первый класс, который наиболее похож на Delphi, но и остальные также имеют право на существование, тем более, что в Delphi включена только Indy. Франсуа Пьетте не согласился на включение его библиотеки в Delphi.

К сожалению, от версии к версии код становится все более и более монстроидальным.

О чем же эта книга, если вы подумали, как следует из названия, что про Indy, то это далеко не так. Эта не книга по Indy, а книга про Интернет, про протоколы, термины, методы работы, а к Indy относятся только примеры. Особенно отличается глава 20, в которой приведены примеры миграции на Delphi for .NET

По окончанию перевод было обнаружено много ошибок и благодаря любезной помощи Михаила Евтеева (Mike Evteev) была проведена серьезная корректировка книги. Мой корявый язык был заменен литературным, уточнена терминология и были исправлены некоторые грамматические ошибки.

Все, кто участвовал в данной работе, являются полноправными членами команды по переводу данной книги.

1. Введение

Добро пожаловать в книгу «Погружение в Indy»

Авторы:

- **Чад Хувер (Chad Z. Hower a.k.a Kudzu)** – Автор Indy и текущий координатор проекта Indy
- **Хади Харири (Hadi Hariri)** – помощник координатора проекта Indy

Данный материал основан на материалах разработчиков и представлен на конференциях разработчиков из нескольких стран, а также материалов разработчиков в различных журналах.

1.1. Об этой книге

Это предварительное издание. В работе еще находится множество материалов. Оно включает объединение, преобразование и импортирование из разных источников. В дополнение, вы увидите несколько глав, с примечаниями, похожими на лепет. Поскольку мы еще пока работаем над материалом. Не волнуйтесь, если некоторые главы пока немного бессмысленны в данное время.

Мы надеемся выпускать обновления раз в месяц в течение первых нескольких месяцев, по мере появления.

Спасибо за вашу поддержку и терпение.

1.1.1. Обратная связь

Пожалуйста, посылайте ваши замечания по данной книге на Indy@atozedsoftware.com. Не обращайтесь за технической поддержкой на данный адрес. Для технической поддержки смотрите главу 2. Техническая поддержка.

1.1.2. Обновления

Обновления можно найти на сайте http://www.atozedsoftware.com/.

1.1.3. Примеры

Все примеры, на которые есть ссылки в книге, доступны на сайте http://www.atozedsoftware.com.

1.2. Дополнительная информация

1.2.1. Другие ресурсы

- **Главный Интернет сайт проекта** - http://www.Indyproject.org/
- **Зеркала проекта** - http://Indy.torry.NET/
- **Портал Indy** – Подписка на новости, статьи, загрузка примеров. http://www.atozedsoftware.com/Indy/
- **Kudzu World - Chad Z. Hower уголок в Сети**. http://www.hower.org/Kudzu/

1.2.2. Для дальнейшего чтения

Данные книги содержат статьи об Indy.

- **Building Kylix Applications** by Cary Jensen

- **Mastering Delphi** by Marco Cantu
- **SDGN Magazine** - SDGN Magazine, официальный журнал SDGN, в котором есть постоянная колонка - The *Indy Pit Stop* written by Chad Z. Hower a.k.a Kudzu.

1.3. Благодарности

Большая благодарность следующим людям, группам и компаниями:

- **Indy Pit Crew и Mercury Team** за многие часы посвященными ими Indy.
- Что бы не забыть, многих людей, которые помогали нам с Indy, чьи имена были забыты.
- **ECSoftware** за разработку справки и руководство, за продукт Help and Manual, с помощью которого и был написан этот документ http://www.helpandmanual.com

2. Техническая поддержка

2.1. Примечание

Пожалуйста, не посылайте письма напрямую членам Indy Pit Crew, если только они не попросят об этом специально. Обычно, когда члены Indy Pit Crew нуждаются в дополнительной информации или в ответ на письмо от члена Indy Pit Crew.

Члены Indy Pit Crew проводят много времени в группах новостей, в тоже время, зарабатывая на жизнь, поэтому они не могут отвечать на технические вопросы через письма. Технические вопросы лучше задавать в соответствующей группе новостей.

Если вам требуется приоритетное обслуживание ваших проблем, то воспользуйтесь платной поддержкой. Даже с платной поддержкой Indy полностью бесплатен, это просто дополнительная возможность.

Если вы до сих пор не поняли, почему вы не должны посылать вопросы напрямую через почту, то посмотрите следующее:

- **Indy делается в команде.** Это означает, что только конкретные люди в состоянии ответственны за конкретные части кода. Посылая письмо, вы вероятнее всего посылаете его не тому человеку, который в состоянии ответить вам быстро.
- **Email поступает только одному человеку** и загружает только этого человека. Посылая сообщение в группу новостей, оно становится доступным, как всем членам Indy Pit Crew, так и другим пользователям Indy, каждый из которых может ответить вам. В дополнение вы можете получить различные ответы от разных людей, которые в состоянии ответить вам.
- **Так же прочитайте статью** How To Ask Questions The Smart Way.

  От переводчика, на моей странице есть один из переводов этой статьи на русском языке - Как правильно задавать вопросы

2.2. Бесплатная поддержка

Бесплатная поддержка может быть получена через группы новостей Borland.

- borland.public.delphi.internet.winsock
- borland.public.cppbuilder.internet.socket
- borland.public.kylix.internet.sockets

Все группы новостей расположены на newsgroups.borland.com. Если у вас нет доступа до групп по NNTP, то вы можете читать их через web интерфейс на http://newsgroups.borland.com. Команда Indy просматривает эти группы регулярно и старается помочь.

2.3. Платная, приоритетная поддержка

Open source software это фантастика, но часто очень плохая поддержка. Indy Experts Support решает данную проблему. Сервис Indy Experts Support обеспечивает вашу компанию приоритетной поддержкой через e-mail или телефон, при достаточно низкой стоимости.

Сервис Indy Experts Support включает:

- Приоритетная поддержка по e-mail

- Поддержка по телефону
- Прямой доступ до авторов и лидеров проекта
- Небольшие куски кода и небольшие проекты
- Приоритетное исправление ошибок и анализ кода
- Консультации через сервис Indy Consulting

Более подробная информация об Indy Experts находится на странице http://www.atozedsoftware.com/Indy/support/.

Сервис Indy Experts Support обслуживается наиболее грамотными экспертами Indy:

- **Chad Z. Hower** – Автор и лидер проекта Indy
- **Hadi Hariri** – Помощник лидера
- **Doychin Bondzhev** – Видный член команды Indy Core Team

2.4. SSL поддержка

Intellicom обслуживает SSL поддержку форумов.

2.5. Отчеты об ошибках

Все сообщения об ошибках должны посылаться через форму сообщения об ошибке на сайте Indy http://www.nevrona.com/Indy or http://Indy.torry.NET/

2.6. Winsock 2

Indy 9.0 требует установленного Winsock 2.0, Windows 98, Windows NT и Windows 2000 имеют установленный Winsock 2.0 по умолчанию. При запуске Indy проверяет версию Winsock.

Windows 95 единственная операционная система, которая не содержит Winsock 2.0. Но его можно установить, если загрузить обновление с сайта Microsoft. Адрес для загрузки следующий: http://www.microsoft.com/windows95/downloads/contents/wuadmintools/s_wunetworkingtools/w95sockets2/

Патч требуется установить в любом случае, поскольку Winsock, который поставляется с Windows 95 имеет серьезные ошибки и утечки памяти, данный патч их устраняет. Windows 95 в данный момент уже редко распространен, менее 5% от общего количества от установленных систем. Еще меньшее количество из них подключено к Интернет, что не вызовет проблем при распространении ваших приложений.

3. Введение в сокеты

3.1. Обзор

Данная глава является введением в концепцию сокетов (TCP/IP сокеты). Это не означает, что мы рассмотрим все аспекты сокетов; это только начальное обучение читателя, что бы можно было начать программировать.

Есть несколько концепций, которые должны быть объяснены в первую очередь. При возможности, концепции будут подобны концепция телефонных систем.

3.2. Стек протоколов TCP/IP

TCP/IP это сокращение от Transmission Control Protocol and Internet Protocol.

TCP/IP может означать разные вещи. Очень часто в общем как слово «хватать все» (*catch all*). В большинстве других случаев, это относится к сетевому протоколу само по себе.

3.3. Клиент

Клиент – это процесс, который инициализирует соединение. Обычно, клиенты общаются с одним сервером за раз. Если процессу требуется общаться с несколькими серверами, то создаются несколько клиентов.

Подобно, телефонному вызову, клиент это та персона, которая делает вызов.

3.4. Сервер

Сервер – это процесс, который отвечает на входящий запрос. Обычно сервер обслуживает некоторое количество запросов, от нескольких клиентов одновременно. Тем не менее, каждое соединение от сервера к клиенту является отдельным сокетом.

Подобно, телефонному вызову, сервер это та персона, которая отвечает на звонок. Сервер обычно устроен так, что в состоянии отвечать на несколько телефонных звонков. Это подобно звонку в центр обслуживания, который может иметь сотен операторов и обслуживание передается первому свободному оператору.

3.5. IP адрес

Каждый компьютер в TCP/IP сети имеет свой уникальный адрес. Некоторые компьютеры могут иметь более одного адреса. IP адрес - это 32-битный номер и обычно представляется с помощью точечной нотации, например 192.168.0.1. Каждая секция представляет собой одни байт 32-битного адреса. IP адрес подобен телефонному номеру. Тем не менее, точно так же, как и в жизни, вы можете иметь более одного телефонного номера, вы можете иметь и более одного IP адрес, назначенного вам. Машины, которые имеют более одного IP адреса, называются multi-homed.

Для разговора с кем-то, в определенном месте, делается попытка соединения (делается набор номера). Ответная сторона, услышав звонок, отвечает на него.

Часто IP адрес для сокращения называют просто IP.

3.6. Порт

Порт – это целочисленный номер, который идентифицирует, с каким приложением или сервисом клиента будет соединение на обслуживание по данному IP адресу.

Порт подобен расширению телефонного номера. Набрав телефонный номер, вы подсоединяетесь к клиенту, но в TCP/IP каждый клиент имеет расширение. Не существует расширений по умолчанию, как в случае с локальной телефонной станцией. В дополнении к IP адресу вы обязаны явно указать порт при соединении с сервером.

Когда серверное приложение готово воспринимать входящие запросы, то оно начинает прослушивать порт. Поэтому приложение или протокол используют общие заранее известные мировые порты. Когда клиент желает пообщаться с сервером, он обязан знать, где приложение (IP адрес/телефонный номер) и какой порт (расширение телефонного номера), приложение прослушивает (отвечает).

Обычно приложения имеют фиксированный номер, чтобы не было проблем для приложения. Например, HTTP использует порт 80, а FTP использует порт 21. Поэтому достаточно знать адрес компьютера, чтобы просмотреть web страницу.

Порты ниже 1024 резервированы и должны использовать только для реализации известных протоколов, которые используют подобный порт для его использования. Большинство популярных протоколов используют резервированные номера портов.

3.7. Протокол

Слово *протокол* происходит от греческого слова protocollon. Это страница, которая приклеивалась к манускрипту, описывающая его содержимое.

В терминах TCP/IP, протокол это описание как производить некоторые действия. НО в большинстве случаев это обычно одна из двух вещей:

1. Тип сокета.
2. Протокол более высокого командного уровня.

Когда говорим о сокетах, то протокол описывает его тип. Распространенные типы сокетов следующие - TCP, UDP и ICMP

Когда говорим о протоколах более высокого уровня, то это относится к командам и ответам, для реализации требуемых функций. Эти протоколы описаны в RFC. Примеры таких протоколов – это HTTP, FTP и SMTP.

3.8. Сокет

Все что здесь говорится об сокетах, относится к TCP/IP. Сокет это комбинация IP адреса, порта и протокола. Сокет также виртуальный коммуникационный трубопровод между двумя процессами. Эти процессы могут быть локальными (расположенными на одном и том же компьютере) или удаленными.

Сокет подобен телефонному соединению, которое обеспечивает разговор. Для того чтобы осуществить разговор, вы обязаны в первую очередь сделать вызов, получить ответ с другой стороны; другими словами нет соединения (сокета) - нет разговора.

3.9. Имя узла

Имя узла это человеческий синоним, вместо IP адреса. Например, есть узел www.nevrona.com. Каждый узел имеет свой эквивалент в виде IP адреса. Для www.nevrona.com это 208.225.207.130.

Использование имен узлов проще для человека, к тому же позволяет сменить IP адрес без проблем для потенциальных клиентов, без их потери.

Имя узла подобно имени человека или названию фирмы. Человек или фирма могут сменить свой телефонный номер, но вы сможете продолжать связываться с ними.

От переводчика: видимо тут намекают на телефонный справочник, как аналог DNS. Иначе, о какой возможности можно говорить.

3.10. Сервис DNS

DNS это сокращение от Domain Name Service.

Задача DNS преобразовывать имена узлов в IP адреса. Для установки соединения, требуется IP адрес, DNS используется, чтобы сначала преобразовать имя в IP адрес.

Что бы сделать телефонный звонок, вы должны набрать телефонный номер. Вы не можете для этого использовать его имя. Если вы не знаете номер человека или если он был изменен, то вы можете посмотреть его номер в телефонной книге. DNS является аналогом телефонной книги.

3.11. Протокол TCP

TCP это сокращение от Transmission Control Protocol.

Иногда TCP также называют потоковым протоколом. TCP/IP включает много протоколов и множество путей для коммуникации. Наиболее часто используемые транспорты это TCP и UDP. TCP это протокол, основанный на соединении, вы должны соединиться с сервером, прежде чем сможете передавать данные. TCP также гарантирует доставку и точность передачи данных. TCP также гарантирует, что данные будут приняты в том же порядке, как и переданы. Большинство вещей, которые используют TCP/IP - используют TCP как транспорт.

TCP соединения, подобны телефонному звонку для разговора.

3.12. Протокол UDP

UDP это сокращение от User Datagram Protocol.

UDP предназначен для датаграмм, и он не требует соединения. UDP позволяет посылать облегченные пакеты на узел без установки соединения. Для UDP пакетов не гарантируется доставка и последовательность доставки. При передаче UDP пакетов, они отсылаются в блоке. Поэтому вы не должны превышать максимальный размер пакета, указанный в вашем TCP/IP стеке.

Поэтому многие люди считают UDP малоприменим. Но это не так, многие потоковые протоколы, такие как RealAudio, используют UDP.

Примечание: термин *потоковый (streaming)* может быть перепутан с термином *потоковое соединение (stream connection)*, которое относится к TCP. Когда вы видите эти термины, вы должны определить, что именно имеется в виду.

Надежность/достоверность UDP пакетов зависит надежности и перегрузки сети. UDP пакеты часто используются в локальных сетях (LAN), поскольку локальная сеть очень надежная и не перегруженная. UDP пакеты, проходящие через Интернет так же обычно надежны и могут использовать коррекцию ошибок передачи или интерполяцию. Доставка не может быть гарантирована в любой сети – потом не будем считать, что ваши данные всегда достигнут точки назначения.

Поскольку UDP не имеет средств подтверждения доставки, то вообще нет гарантии этой доставки. Если вы посылаете UDP пакет на другой узел, то вы не имеете возможности узнать, доставлен пакет или нет. Стек не может определить это и не выдает никакой информации об

ошибке, если пакет не доставлен. Если вам требуется подобная гарантия, то вы должны сами организовать ответ от удаленного узла.

UDP аналогичен посылке сообщения на обычный пейджер. Вы знаете, что вы послали, но вы не знаете получено ли оно. Пейджер может не существовать, или находиться вне зоны приема, или может быть выключен, или не работоспособен, в дополнение сеть может потерять ваше сообщение. Пока обратная сторона не сообщит о приеме сообщения, вы этого не узнаете. В дополнение, если вы посылаете несколько сообщений, то возможно они поступят совсем в другом порядке.

Дополнительная информация по UDP находится в главе 7.

3.13. Протокол ICMP

ICMP это сокращение от Internet Control Message Protocol.

ICMP это протокол управления и обслуживания. Обычно, вам не потребуется использовать этот протокол. Обычно он используется для общения с маршрутизаторами и другим сетевым оборудованием. ICMP позволяет узлам получать статус и информацию об ошибке. ICMP используется для протоколов PING, TRACEROUTE и других подобных.

3.14. Файл HOSTS

Файл HOSTS это текстовый файл, который содержит локальную информацию об узлах.

Когда стек пытается определить IP адрес для узла, то он сначала просматривает этот файл. Если информация будет найдена, то используется она, если же нет, то процесс продолжается с использованием DNS.

Это пример файла HOSTS:

```
# This is a sample HOSTS file
Caesar   192.168.0.4  # Server computer
augustus 192.168.0.5  # Firewall computer
```

Имя узла и IP адрес должны быть разделены с помощью пробела или табуляции. Комментарии также могут быть вставлены в файл, для этого должен быть использован символ #.

Файл HOSTS может быть использован для ввода ложных значений или для замены значений DNS. Файл HOSTS часто используется в малых сетях, которые не имеют DNS сервера. Файл HOSTS также используется для перекрытия IP адресов при отладке. Вам не требуется читать этот файл, поскольку стек протоколов делает это самостоятельно и прозрачно для вас.

3.15. Файл SERVICES

Файл SERVICES подобен файлу HOSTS. Вместо разрешения узлов в IP адреса, он разрешает имена сервисов в номера портов.

Ниже пример, урезанный, файла SERVICES. Для полного списка сервисов вы можете обратиться к RFC 1700. RFC 1700 содержит определение сервисов и их портов:

```
Echo     7/tcp
Echo     7/udp
Discard  9/tcp       sink null
Discard  9/udp       sink null
Systat   11/tcp      users             #Active users
Systat   11/tcp      users             #Active users
Daytime  13/tcp
Daytime  13/udp
```

```
Qotd     17/tcp        quote             #Quote of the day
qotd     17/udp        quote             #Quote of the day
chargen  19/tcp        ttytst source     #Character generator
chargen  19/udp        ttytst source     #Character generator
ftp-data 20/tcp                          #FTP data
ftp      21/tcp                          #FTP control
telnet   23/tcp
smtp     25/tcp                          mail #Simple Mail Transfer Protocol
```

Формат файла следующий:

```
<service name> <port number>/<protocol> [aliases...] [#<comment>]
```

Вам не требуется читать данный файл, поскольку стек протоколов делает это самостоятельно и прозрачно для вас. Файл SERVICES может быть прочитан с помощью специальной функции стека, но большинство программ не используют эту функции и игнорируют их значения. Например, многие FTP программы используют порт по умолчанию без обращения к функциям стека, для определения номера порта по имени ‘FTP’.

Обычно вы никогда не должны изменять этот файл. Некоторые программы, тем не менее, добавляют свои значения в него и реально используют его. Вы можете изменить их значения, чтобы заставить использовать программу другой порт. Одна из таких программ – это Interbase. Interbase добавляет в файл следующую строку:

```
gds_db3050/tcp
```

Вы можете изменить ее и Interbase будет использовать другой порт. Обычно это не слишком хорошая практика делать подобное. Но это может потребоваться, если вы пишите приложение с сокетами, обычно серверное приложение. Так же хорошей практикой при написании клиентов – это использование функций стека, для получения значений из файла SERVICES, особенно для нестандартных протоколов. Если вхождение не найдено, то можно использовать порт по умолчанию.

3.16. Localhost (*Loopback*)

LOCALHOST подобен "Self" в Delphi или "this" в C++. LOCALHOST ссылается на компьютер, на котором выполняется приложение. Это адрес обратной петли и это реальный физический IP адрес, со значением 127.0.0.1. Если 127.0.0.1 используется на клиенте, он всегда возвращает пакет обратно на тот же компьютер, для сервера на том же компьютере, что и клиент.

Это очень полезно для отладки. Это также может быть использовано для связи с сервисами, запущенными на этом же компьютере. Если вы имеете локальный web сервер, то вам не надо знать его адрес и изменять свои скрипты, каждый раз как адрес будет изменен, вместо этого используйте 127.0.0.1.

От переводчика: вообще то Localhost, может иметь и другой адрес, но как правило, это 127.0.0.1

3.17. Программа Ping

Ping - это протокол, который проверяет доступен ли узел с локального компьютера, или нет. Ping обычно используется в диагностических целях.

Ping работает из командной строки, синтаксис использования следующий:

```
ping <host name or IP>
```

Если узел доступен, то вывод выглядит так:

```
C:\ping localhost

Обмен пакетами с xp.host.ru [127.0.0.1] по 32 байт:

Ответ от 127.0.0.1: число байт=32 время<1мс TTL=128
Ответ от 127.0.0.1: число байт=32 время<1мс TTL=128
Ответ от 127.0.0.1: число байт=32 время<1мс TTL=128
Ответ от 127.0.0.1: число байт=32 время<1мс TTL=128

Статистика Ping для 127.0.0.1:
    Пакетов: отправлено = 4, получено = 4, потеряно = 0 (0% потерь
Приблизительное время приема-передачи в мс:
    Минимальное = 0мсек, Максимальное = 0 мсек, Среднее = 0 мсек
```

Если узел не доступен, то вывод выглядит так:

```
C:\>ping 192.168.0.200

Pinging 192.168.0.200 with 32 bytes of data:

Request timed out.
Request timed out.
Request timed out.
Request timed out.

Ping statistics for 192.168.0.200:
    Packets: Sent = 4, Received = 0, Lost = 4 (100% loss),
```

3.18. Программа TraceRoute

TCP/IP пакеты не двигаются напрямую от узла к узлу. Пакеты маршрутизируются подобно движению автомобилей от одного дома до другого. Обычно, автомобиль должен двигаться более чем по одной дороге, пока не достигнет точки назначения. TCP/IP пакеты двигаются подобным образом. Каждый раз пакет сменяет «дорогу» от маршрутизатора (*node*) к маршрутизатору. Получив список маршрутизаторов можно определить список узлов (*host*), по которым путешествует пакет. Это очень полезно для диагностики, почему тот или другой узел недоступен.

Traceroute работает из командной строки, Traceroute показывает список IP маршрутизаторов, через которые проходит трасса до узла назначения и сколько требуется времени на каждый прыжок, до каждого узла. Данное время пригодна для поиска узких мест. Traceroute показывает последний маршрутизатор, который нормально обработал пакет, в случае неудачной передачи. Traceroute используется для дальнейшей диагностики после пинга.

Пример вывода работы Traceroute при удачном прохождении:

```
C:\>tracert www.atozedsoftware.com

Tracing route to www.atozedsoftware.com [213.239.44.103]
over a maximum of 30 hops:

1 <1 ms <1 ms <1 ms server.mshome.NET [192.168.0.1]
2 54 ms 54 ms 50 ms 102.111.0.13
3 54 ms 51 ms 53 ms 192.168.0.9
4 55 ms 54 ms 54 ms 192.168.5.2
5 55 ms 232 ms 53 ms 195.14.128.42
6 56 ms 55 ms 54 ms cosmos-e.cytanet.NET [195.14.157.1]
7 239 ms 237 ms 237 ms ds3-6-0-cr02.nyc01.pccwbtn.NET [63.218.9.1]
8 304 ms 304 ms 303 ms ge-4-2-cr02.ldn01.pccwbtn.NET [63.218.12.66]
9 304 ms 307 ms 307 ms linx.uk2net.com [195.66.224.19]
10 309 ms 302 ms 306 ms gw12k-hex.uk2net.com [213.239.57.1]
```

```
11 307 ms 306 ms 305 ms pop3 [213.239.44.103]

Trace complete.
```

3.19. LAN

LAN это сокращение от *Local Area Network*.

Что именно локальная сеть очень зависит от конкретной топологии сети и может варьироваться. Тем не менее, LAN относится ко всем системам, подключенным к Ethernet повторителям (*hubs*) и коммутаторам (*switches*), или в некоторых случаях к Token ring или другим. К LAN не относятся другие LAN, подключенные с помощью мостов или маршрутизаторов. То есть к LAN относятся только те части, до которых сетевой трафик доходит без использования мостов и маршрутизаторов.

Можно думать об LAN, как об улицах, с мостами и маршрутизаторами, которые объединяют города скоростными трассами.

3.20. WAN

WAN это сокращение от *Wide Area Network*.

WAN означает соединение нескольких LAN совместно, с помощью мостов и маршрутизаторов в одну большую сеть.

Используя пример с городом, WAN состоит из множества городов (LAN) соединенных скоростными трассами. Интернет сам по себе классифицируются как WAN.

3.21. IETF

IETF (Internet Engineering Task Force) это открытое сообщество, которое продвигает функционирование, стабильность и развитие Интернет. IETF работает подобно Open Source разработке программ. IETF доступен на сайте http://www.ietf.org/.

3.22. RFC

RFC это сокращение от Request for Comments.

RFC это набор официальных документов от IETF, которые описывают и детализируют протоколы Интернет. Документы RFC идентифицируются их номера, подобными RFC 822.

Есть очень много зеркал, которые содержат документы RFC в Интернет. Лучший из них, который имеет поисковую системе находится на сайте http://www.rfc-editor.org/

RFC редактор (web сайт указанный выше) описывает документы RFC как:

> *Серия документов* RFC – это набор технических и организационных заметок об Интернет (изначально *ARPANET), начиная с 1969 года. Заметки в серии RFC документов дискутируют многие аспекты компьютерных сетей, включая протоколы, процедуры, программы и концепции, как заметки, мнения, а иногда и юмор.*

3.23. Кодовые потоки (*thread*)

Кодовые потоки – это метод выполнения программы. Большинство программ имеют только один поток. Тем не менее, дополнительные потоки могут быть созданы для выполнения параллельных вычислений.

На системах с несколькими CPU, потоки могут выполняться одновременно на разных CPU, для более быстрого выполнения.

На системах с одним CPU, множество потоков могут выполняться с помощью вытесняющей многозадачности. При вытесняющей многозадачности, каждому потоку выделяется небольшой квант времени. Так что, кажется, что каждый поток выполняется на отдельном процессоре.

3.24. Fork

Unix до сих пор не имеет поддержки потоков. Вместо этого, Unix использует ветвление (*forking*). С потоками, каждая отдельная строка выполнения исполняется, но она существует в том же самом процессе, как и другие потоки и в том же адресном пространстве. При разветвлении каждый процесс должен сам себя разделять. Создается новый процесс и все хендлы (*handles*) передаются ему.

Разветвление не так эффективно как потоки, но также имеется и преимущества. Разветвление более стабильно. В большинстве случаев - разветвление легче программировать.

Разветвление типично для Unix, так как ядро использует и поддерживает его, в то время как потоки это более новое.

3.25. Winsock

Winsock – это сокращение от *Windows Sockets*.

Winsock – это определенное и документированное стандартное API, для программирования сетевых протоколов. В основном используется для программирования TCP/IP, но может быть использовано и для программирования Novell (IPX/SPX) и других сетевых протоколов. Winsock реализован как набор DLL и является частью Win32.

3.26. Стек протоколов

Термин стек протоколов относится к слою операционной системы, которая сеть и предоставляет API для разработчика по доступу к сети.

В Windows стек протоколов реализован с помощью Winsock.

3.27. Сетевой порядок байт

Различные компьютерные системы хранят числовые данные в различном порядке. Некоторые компьютеры хранят числа, начиная с самого наименее значимого байта (*LSB*), тогда как другие с наиболее значимого байта (*MSB*). В случае сети, не всегда известно, какой компьютер используется на другой стороне. Для решения этой проблемы был принят стандартный порядок байт для записи и передачи по сети, названый сетевой порядок байт. Сетевой порядок байт это фиксированный порядок байт, который должен использоваться в приложении при передаче двоичных чисел.

4. Введение в Indy

4.1. Путь Indy

Indy разработан изначально на использование потоков. Построение серверов и клиентов в Indy подобно построению серверов и клиентов в Unix, исключая, что это много проще, поскольку у вас есть Indy и Delphi. Приложения в Unix обычно вызывают стек напрямую с минимальным уровнем абстракции или вообще без него.

Обычно сервера в Unix имеют один или несколько слушающих процессов, которые наблюдают за пользовательскими запросами. Для каждого клиента, которого требуется обслужить, создается разветвление (*fork*) процесса. Это делает программирование очень простым, так как каждый процесс обслуживает только одного клиента. Процесс так запускается в собственном контексте безопасности, который может быть установлен на основе слушателя, правах, аутентификации или других предпосылках.

Сервера Indy работают подобным образом. Windows в отличии от Unix, не делает разветвление, а создает новый поток. Сервера Indy создают новый поток для каждого клиентского соединения. Сервера Indy создают слушающий поток, который изолирован от главного кодового потока программы. Слушающий поток случает входящие клиентские запросы. Для каждого клиента, которому отвечают, создается новый поток для его обслуживания. Необходимые события возбуждаются в контексте данного потока.

4.2. Методология Indy

Indy отличается от других сокетных компонент, с которыми вы возможно уже знакомы. Если вы никогда не работали с другими сокетными компонентами, возможно, вы найдете, что Indy очень прост, так как Indy работает так как вы ожидали. Если вы уже работали с другими сокетными компонентами, то просто забудьте все, что вы знали. Это будет вам только мешать и вы будете делать ложные предпосылки.

Почти все другие компоненты работают в неблокирующем режиме, асинхронно. Они требуют от вас реагировать на события, создавать машину состояний и часто исполнять циклы ожидания. Например, с другими компонентами, когда вы делаете соединения, то вы должны ожидать событие соединения или крутить цикл ожидания, пока свойство, ухаживающие факт соединение не будет установлено. С Indy, вы просто вызываете метод Connect и просто ждете возврата из него. Если соединение будет успешное, то будет возврат из метода по окончанию соединения. Если же соединение не произойдет, то будет возбуждено исключение.

Работа с Indy аналогична работе с файлами. Indy позволяет поместить весь код в одно место, вместо создания различных разработчиков событий. В дополнение, многие находят Indy более простым в использовании. Indy также разработан на работу с потоками. Если вы имеет проблемы с реализацией чего-либо в Indy, то вернитесь назад и реализуйте это как для файлов.

4.3. Различия Indy

- Indy использует API блокирующих сокетов.
- Indy не ориентирован на события. Indy имеет события, но для информационных нужд, но они не обязательны.
- Indy разработан ни использование кодовых потоков. Тем не менее, Indy может работать без использования потоков.

- Программирование в Indy – это линейное программирование.
- Indy имеет высокий уровень абстрагирования. Большинство сокет компонент не очень эффективно изолируют программиста от стека. Большинство сокет компонент вместо изоляции от стека, наоборот погружают его в сложности создания оберток вокруг этого в Delphi / C++ Builder.

4.4. Обзор клиентов

Indy разработан для создания высокого уровня абстракции. Сложности и детализация TCP/IP стека скрыты от программиста.

Типичный клиент сессия в Indy выглядит так:

```
with IndyClient do begin
  Host := 'postcodes.atozedsoftware.com'; // Host to call
  Port := 6000; // Port to call the server on
  Connect;
  try
    // Do your communication
  finally
    Disconnect;
  end;
end;
```

4.5. Обзор серверов

Компоненты серверов Indy создают слушающий поток, который изолирован от главного кодового потока программы. Слушающий поток случает входящие клиентские запросы. Для каждого клиента, которому отвечают, создается новый поток для его обслуживания. Необходимые события возбуждаются в контексте данного потока.

4.6. Потоки

Для реализации функциональности используются потоки. Indy очень интенсивно использует потоки для реализации серверов, потоки так же используются и в клиентах. неблокирующие сокеты также могут использовать потоки, но они требуют некоторой дополнительной обработки и их преимущества теряются по сравнению блокирующих сокетов.

5. Блокирующий режим против неблокирующего

5.1. Модели программирования

В Windows есть две модели программирования сокетов – блокирующий и неблокирующий. Иногда они также называются как – синхронный (*blocking*) и асинхронный (*non-blocking*). В данном документе мы будем использовать термины блокирующий и неблокирующий.

На платформе Unix, поддержан только блокирующий режим.

5.2. Другие модели

В действительности есть еще несколько реализованных моделей. Это completion ports, and overlapped I/O. Но использование этих моделей требует гораздо больше кода и обычно используется только в очень сложных серверных приложениях.

В дополнение, данные модели не кросс платформенные и их реализация сильно отличает от одной операционной системы к другой.

Indy 10 содержит поддержку и этих моделей.

5.3. Блокирующий режим

В Indy используются вызовы блокирующих сокетов. Блокирующие вызовы очень похожи на чтение/запись файлов. Когда вы читаете файл или пишите файл, то возврат из функции не происходит до ее окончания. Различие состоит в том, что обычно требуется значительно больше времени до окончания. Операции чтения и записи зависят от скорости сети.

С Indy, вы просто вызываете метод Connect и просто ждете возврата из него. Если соединение будет успешное, то будет возврат из метода по окончанию соединения. Если же соединение не произойдет, то будет возбуждено исключение.

5.4. Неблокирующий режим

Работа неблокирующих сокетов основана на системных событиях. После того как произведен вызов, будет возбуждено событие.

Например, для попытки соединения сокета, вы должны вызвать метод Connect. Данный метод немедленно возвращает управление в программу. Когда сокет будет подсоединен, то будет возбуждено событие. Это требует, что бы логика связи была разделена по многим процедурам или использовать циклы опроса.

5.5. История Winsock

В начале был Unix. Это был Berkely Unix. Он имел стандартное API для поддержки сокетов. Это API было адоптировано в большинстве Unix систем.

Затем появился Windows, и кто-то посчитал, что это хорошая идея иметь возможность программировать TCP/IP и в Windows. Поэтому они портировали API Unix сокетов. Это позволило большинство Unix кода с легкостью портировать и в Windows.

5.6. Блокирующий режим это не смертельно

Из-за блокирующего режима мы неоднократно были биты нашими противниками, но блокирующий режим не является дьяволом.

Когда API Unix сокетов было портировано в Windows, то ему дали имя Winsock. Это сокращение от "Windows Sockets".

В Юниксе типично проблема решалась за счет разветвления (похоже на много поточность, но за счет отдельных процессов вместо потоков). Юникс клиенты и демоны (*daemons*) должны были раздваивать процессы для каждого сокета. Данные процессы затем выполнялись независимо и использовали блокирующие сокеты.

Windows 3.x не мог распараллеливаться и плохо поддерживал многозадачность. Windows 3.1 также не имел поддержки потоков. Использование блокирующих сокетов замораживало пользовательский интерфейс и делало программы не реагирующими. Поскольку это было не приемлемо, то к WinSock были добавлены неблокирующие сокеты, позволяя Windows 3.x с его ограничениями использовать Winsock без замораживания всей системы. Это потребовало другого программирования сокетов, Microsoft и другие страстно поносили блокирующие режимы, что бы скрыть недостатки Windows 3.x.

Затем пришли Windows NT и Windows 95, Windows стала поддержать вытесняющую многозадачность и потоки. Но к этому моменту мозги уже были запудрены (то есть разработчики считали блокирующие сокеты порождением дьявола), и уже было тяжело изменить содеянное. По этому поношение блокирующих режимов продолжается.

В действительности, блокирующее API единственное которое поддерживает Unix.

Некоторые расширения, для поддержки неблокирующих сокетов были добавлены и в Unix. Эти расширения работают совсем не так как в Windows. Эти расширения не стандартны для всех Unix платформ и не используются широко. Блокирующие сокеты в Unix все еще используются в каждом приложении и будут продолжаться использоваться и дальше.

Блокирующие сокеты также имеют и другие преимущества. Блокирующие сокеты много лучше для поточности, безопасности и по другим аспектам.

5.7. Достоинства блокирующего режима

1. **Проще программировать** - Блокирующие сокеты проще программировать. Весь пользовательский код может находиться в одном месте и выполняться в естественном, последовательном порядке.
2. **Кросс-платформенность** – поскольку Unix использует блокирующие сокеты, по переносимый код легче писать. Indy использует данный факт для использования своего кода между платформами. Другие сокет компоненты, которые кросс платформенные, на самом деле эмулируют это с помощью внутреннего вызова блокирующих сокетов.
3. **Удобнее работать с потоками** - Поскольку у блокирующих сокетов последовательность приобретена по наследственности, поэтому их очень просто использовать в потоках.
4. **Независимость от сообщений** – неблокирующие сокеты зависят от системы оконных сообщений. Когда используются потоки, то создается отдельная очередь сообщений. Но когда потоки не используются, то узким местом становится обработка множества соединений.

5.8. Недостатки блокирующего режима

1. **Пользовательский интерфейс замораживается в клиентах** - Вызов блокирующего сокета не возвращает управления, пока не выполнит свою задачу. Когда подобные вызовы делаются в главном кодовом потоке, то приложение замораживает пользовательский интерфейс. Замораживание происходит, поскольку сообщения обновления, перерисовки и другие сообщения не обрабатываются до окончания вызова блокирующего сокета.

5.9. Компонент TIdAntiFreeze

В Indy имеется специальный компонент, который решает проблему замораживания пользовательского интерфейса. Просто добавьте один компонент TIdAntiFreeze куда ни будь в своем приложении, и вы сможете выполнять блокирующие вызовы без замораживания пользовательского интерфейса. Сам компонент будет рассмотрен в подробностях чуть позже.

Использование TIdAntiFreeze позволяет получить все преимущества блокирующих сокетов, без видимых недостатков.

5.10. Достоинства неблокирующего режима

1. **Пользовательский интерфейс не замораживается** – поскольку пользовательский код обрабатывает оконные сообщения, то имеет контроль и над сокетными сообщениями. Поэтому Windows также может обрабатывать и другие сообщения.
2. **Многозадачность без использования потоков** – используется единственный кодовый поток для обработки множества сокетов.
3. **Очень малая нагрузки при множестве сокетов** – поскольку множество сокетов могут обрабатываться без потоков, то нагрузка на память и процессор значительно ниже.

5.11. Недостатки неблокирующего режима

1. **Более сложное программирование** – неблокирующие сокеты требуют использования опроса или обработки событий. События наиболее используемый метод, а циклы опроса менее эффективны. При использовании обработчиков событий, код размазан по куче процедур, поэтому требуется отслеживание состояния. Это означает большее количество ошибок и более сложная модификация кода.

5.12. Сравнение технологий

Если вы хорошо знаете Indy и его методологию, то вы можете пропустить эту главу. Но даже если вы ранее программировали сокеты, до использования Indy, то все равно данная глава будет вам полезна.

Для тех, кто никогда не программировал сокеты до Indy, то будет легко и естественно использовать его. Но для тех кто программировал сокеты ранее, Indy будет камнем преткновения. Поскольку Indy работает совсем по другому. Попытка программировать в Indy тем же самым образом. Это не означает, что другие решения неправильные, просто Indy работает иначе. Пытаться программировать в Indy так же, как с другими сокетными библиотеками, равносильна попытке приготовить пирожное в микроволновой печи, как в духовке. Результатом будет испорченное пирожное.

Если вы использовали другие сокетные библиотеки ранее, пожалуйста следуйте следующему девизу:

Забудьте все, что вы знали раньше!

Это легко сказать, труднее сделать, менять привычки тяжело. Чтобы подчеркнуть разницу, приведем абстрактный пример. Для абстрагирования концепции, используем в качестве аналога файлы. Данный документ подразумевает, что вы умеете работать с файлам. Надеемся, что Бейсик программисты не читают эту книгу.

5.13. Файлы против сокетов

Разница между файлами и сокетами в основном в скорости доступа. Доступ к файлу не всегда быстрый. Флоппи диски, сетевые диска, ленточные устройства архивирования и иерархические системы хранения часто имеют медленную скорость.

5.14. Сценарий записи в файл

Представим простой сценарий записи в файл. Поскольку данная процедура очень простая, то она очень подходит демонстрации.

1. Открыть файл
2. Записать данные
3. Закрыть файл

5.15. Блокирующий режим записи файла

Блокирующая запись в файл выглядит следующим образом:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  s: string;
begin
  s := 'Indy Rules the (Kudzu) World !' + #13#10;
  try
    // Open the file
    with TFileStream.Create('c:\temp\test.dat', fmCreate) do
    try
      // Write data to the file
      WriteBuffer(s[1], Length(s));
```

```
        // Close the file
      finally
        Free;
      end;
    end;
  end;
```

Как вы видите, это практически повторяет приведенный выше псевдокод. Код последовательный и легкий для понимания.

5.16. Неблокирующий режим записи файла

Не существует такой вещи как неблокирующий режим записи файла (может быть исключая overlapped I/O, но это за пределами данной книги), но здесь мы можем просто эмулировать механизм это. File1 это условный неблокирующий компонент, размещенной на форме.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  File1.Filename := 'd:\temp\test.dat';
  File1.Open;
end;

procedure TForm1.File1OnOpen(Sender: TObject);
var
  i: integer;
begin
  FWriteData := 'Hello World!' + #13#10;
  i := File1.Write(FWriteData);
  Delete(FWriteData, 1, i);
end;

procedure TForm1.File1OnWrite(Sender: TObject);
var
  i: integer;
begin
  i := File1.Write(FWriteData);
  Delete(FWriteData, 1, i);
  if Length(FWriteData) = 0 then
  begin
    File1.Close;
  end;
end;

procedure TForm1.File1OnClose(Sender: TObject);
begin
  Button1.Enabled := True;
end;
```

Потратим немного времени, что бы попытаться понять, что здесь делается. Если вы используете неблокирующие сокеты, то вы должны легко понимать данный код. Это примерно следующее:

1. При вызове Button1Click открывается файл. Метод Open немедленно вернет управление в программу, но файл еще не открыт и нельзя с ним еще нельзя работать.
2. Обработчик события OnOpen будет возбужден, когда файл будет открыть и готов к работе. Делается попытка записать данные в файл, но все данные еще не акцептированы. Метод Write вернет количество записанных байт. Оставшиеся данные будут сохранены позже.
3. Обработчик события OnWrite будет возбужден, когда файл будет готов воспринять следующую порцию данных, и метод Write будет повторяться для оставшихся данных.

4. Шаг 3 повторяется до тех пор, пока все данные не будут записаны методом Write. По окончанию записи всех данных вызывается метод Close. Но файл пока еще не закрыт.
5. The OnClose event is fired. The file is now closed.

5.17. Сравнение записи файлов

Оба примера только записывают данные. Чтение и запись данных будут сложнее для неблокирующего режима, но только добавлением одной строки для блокирующего режима.

Для блокирующего примера, просто откройте, записывайте данные, и закройте файл когда необходимо:

- 3 File1 события
- 1 поле в Form

Неблокирующая версия более сложная и более тяжелая для понимания. Дадим шанс выбора между обеими, если надо будет выбирать, то большинство выберет неблокирующий путь. Большинство C++ программистов, исключая конечно просто мазохистов или вообще не будет выбирать, поскольку все они почти просты. Почти все сокетные функции используют неблокирующий режим.

5.18. Почти как файлы

Использование Indy почти равносильно использованию файлов. В действительности Indy еще проще, поскольку Indy имеет ряд методов для чтения и записи. Indy пример, эквивалентный примеру с файлами выглядит так:

```
with IndyClient do
begin
     Connect;
     Try
       WriteLn('Hello World.');
     finally
       Disconnect;
     end;
end;
```

Как вы можете видеть, Indy в действительности очень похож работе с файлами. Метод Connect замещает функцию Open, а метод Disconnect замещает функцию Close. Если вы думаете и признаете сокеты как чтение и запись в файл, то вам будет использовать Indy очень просто.

6. Введение в клиентов

6.1. Базовый клиент

Базовый клиент Indy выглядит так:

```
with IndyClient do
begin
  Host := 'test.atozedsoftware.com';
  Port := 6000;
  Connect;
  Try
    // Read and write data here
  finally
    Disconnect;
  end;
end;
```

host и port могут быть установлены во время разработки с помощью инспектора объектов. Это минимальный код, который требуется при написании клиента в Indy. Минимальные требования для создания клиентов следующие:

1. Установка свойства Host.
2. Установка свойства Port. Требуется, если нет порта по умолчанию. Большинство протоколов имеют такой порт.
3. Соединение.
4. Передача данных. Включает чтение и запись.
5. Разъединение.

6.2. Обработка исключений

Обработка исключений в клиентах Indy такая же как с файлами. Если ошибка возникнет во время выполнения любого метода Indy, то будет возбуждено соответствующее исключение. Для обработки исключения код надо помещать в блоки try..finally или try..except blocks.

Также отсутствует событие OnError, так что не ищите его. Это может показаться странным, если вы уже работали с другими сокетными библиотеками, но посмотрите на TFileStream, он также не имеет события OnError, просто если есть проблема, то возбуждается исключение. Indy работает подобным образом.

Подобно тому, как все открытые файлы должны быть закрыты, все вызовы Connect в Indy должны быть закрытым вызовом метода Disconnect. Базовые клиенты должны начитать работу следующим образом:

```
Client.Connect;
try
  // Perform read/write here
finally
  Client.Disconnect;
end;
```

Исключения Indy только слегка отличаются от исключений VCL, все исключения Indy наследуются от EIdException. Если вы желаете обрабатывать исключения Indy отдельно от исключений VCL, то это можно сделать, как в следующем примере.

Примечание: Для использования EIdException вы должны добавить IdException в uses.

```
try
  Client.Connect;
  try
    // Perform read/write here
  finally
    Client.Disconnect;
  end;
except
  on E: EIdException do
  begin
    ShowMessage('Communication Exception: ' + E.Message);
  end
  else
  begin
    ShowMessage('VCL Exception: ' + E.Message);
  end;
end;
```

Если произойдет ошибка во время вызова метода Connect, то она будет очищена самостоятельно перед возбуждения соответствующего исключения. Поэтому, try здесь после вызова метода Connect на не перед. Тем не менее, если исключение случится во время передачи данных, то будет возбуждено исключение raised. Сокет останется подсоединенным. Это позволяет вам повторить операцию передаче или отсоединиться. В приведенном выше примере, не делается никакой дополнительной обработки и сокет отсоединяется по любой ошибке, и производится нормальное завершение.

Для обработки ошибок во время соединения и отделения от других ошибок связи, требуется изменить ваш код:

```
try
  IdTCPClient1.Connect;
  try
    try
      // Do your communications here
    finally
      IdTCPClient1.Disconnect;
    end;
  except
    on E: EIdException do
    begin
      ShowMessage('An network error occurred during communication: ' +
        E.Message);
    end;
    on E: Exception do
    begin
      ShowMessage('An unknown error occurred during communication: ' +
        E.Message);
    end;
  end;
except
  on E: EIdException do
  begin
    ShowMessage('An network error occurred while trying to connect: ' +
        E.Message);
  end;
  on E: Exception do
  begin
    ShowMessage('An unknown error occurred while trying to connect: ' +
        E.Message);
```

```
    end;
  end;
```

Данный код не только проверяет исключения, которые возникают во время соединения, но и отделяет эти ошибки от других ошибок связи. Дальше исключения Indy изолируются от других исключений.

6.3. Исключения это не ошибки

Многие разработчики серьезно считаю, что исключения это ошибки. Но это не так. Если бы это было так, то Borland бы назвал из ошибками, а не исключениями.

Исключение – это что-то, что за пределами ординарного. В терминах программирования, исключение – это что-то, что прерывает нормальный поток исполнения.

Исключения используются для представления ошибок в Delphi и по этому большинство исключений это ошибки. Тем не менее, есть такие исключения, как EAbort, которое не является ошибкой. Indy также определяет ряд исключений, которые не являются ошибками. Такие исключения, как правило, наследованы от EIdSilentException и могут быть легко отделены от ошибок и других исключений. Более сложный пример можно посмотреть в EIdConnClosedGracefully.

6.4. Компонент TIdAntiFreeze

Indy имеет специальный компонент, который прозрачно разрешает проблему с замораживанием пользовательского интерфейса. Достаточно одного экземпляра компонента TIdAntiFreeze в приложении, позволяя использовать блокирующие вызовы в главном кодовом потоке, без замораживания пользовательского интерфейса.

TIdAntiFreeze работает внутренне, независимо от вызова стека, и позволяет обрабатывать сообщения в течении периода таймаута. Внешний вызовы Indy продолжают быть блокированы и их код работает точно так же, как и без компонента TIdAntiFreeze.

Поскольку пользовательский интерфейс замораживается только при вызове блокирующих сокетов в главном кодовом потоке, TIdAntiFreeze влияет только на вызовы Indy, сделанные из главного кодового потока. Если приложение использует вызовы Indy из других потоков, TIdAntiFreeze не требуется. Но если используется, то влияет на вызовы сделанные только из главного кодового потока.

Использование TIdAntiFreeze немного замедляет работу сокетов. Сколько давать приоритета приложению задается в свойствах TIdAntiFreeze. Причина, по которой TIdAntiFreeze замедляет сокетовые операции, состоит в том, что главному кодовому потоку разрешается обрабатывать сообщения. По этому надо позаботиться, чтобы не позволять много времени отводилось обработке сообщений. Это включает большинство таких событий, как OnClick, OnPaint, OnResize и многие другие. Поскольку неблокирующие сокеты тоже обмениваются сообщениями, этаже проблема относится и к ним. С Indy и с помощью использования TIdAntiFreeze, программист получает полный контроль.

6.5. Пример - Проверка почтового индекса - клиент

Данный пример – это клиент, протокол просмотра почтовых индексов. Протокол очень простой и предполагается, что сервер уже реализован. В данной главе рассматривается только клиент.

Клиент обеспечивает получении имени города и штата по почтовому индексу (Zip код для американских пользователей). Исходные данные находятся на сервере для американских почтовых индексов. Американские почтовые индексы (называемые zip коды) состоят из 5 цифр.

Код сервера будет приведен позже.

6.5.1. Проверка почтового индекса - протокол

Протокол клиента очень прост, он содержит только две команды:

- Lookup <почтовый код 1> < почтовый код 2> ...
- Quit

Общение с сервером выглядит так:

```
Server: 204 Post Code Server Ready.
Client: lookup 16412
Server: 200 Ok
Server: 16412: EDINBORO, PA
Server: .
Client: lookup 37642 77056
Server: 200 Ok
Server: 37642: CHURCH HILL, TN
Server: 77056: HOUSTON, TX
Server: .
Client: quit
Server: 201-Paka!
Server: 201 4 requests processed.
```

The server responds with a greeting when the client connects. Greetings and replies to commands typically contain a 3 digit number specifying status. This will be covered more in detail in later sections.

После приветственного сообщения сервер готов принимать запросы от клиента. Если принята команда Lookup – сервер отвечает списком почтовых кодов и далее соответствующим именем города и штата. Ответ заканчивается строкой с единственным символом <точка>. Клиент может посылать множество команд, пока не выдаст команду Quit, после происходит рассоединение.

6.5.2. Объяснение кода

Клиент просмотра почтового кода содержит две кнопки, listbox и memo. Одна кнопка используется для очистки окна результатов, а другая для получения ответов от сервера и запрос информации от него. Результаты помещаются в listbox.

В обычном приложении, пользователь должен предоставить информацию об узле, порте и возможно о прокси. Но для демонстрации данная информация указана в коде. В качестве узла используется 127.0.0.1 и порт 6000.

Когда пользователь нажимает на кнопку Lookup, то выполняется следующий обработчик:

```
procedure TformMain.butnLookupClick(Sender: TObject);
var
  i: integer;
begin
  butnLookup.Enabled := true;
  try
    lboxResults.Clear;
    with Client do
    begin
```

```
        Connect;
        try
          // Read the welcome message
          GetResponse(204);
          lboxResults.Items.AddStrings(LastCmdResult.Text);
          lboxResults.Items.Add('');
          // Submit each zip code and read the result
          for i := 0 to memoInput.Lines.Count - 1 do
          begin
            SendCmd('Lookup ' + memoInput.Lines[i], 200);
            Capture(lboxResults.Items);
            lboxResults.Items.Add('');
          end;
          SendCmd('Quit', 201);
        finally
          Disconnect;
        end;
      end;
    finally
      butnLookup.Enabled := True;
    end;
  end;
```

Методы Indy, использованные здесь, объясняются только коротко, поскольку подробно они рассмотрены в других главах.

Когда код выполняется, то блокируется кнопка, чтобы пользователь не мог послать другой запрос, пока не закончен текущий. Вы можете подумать, что это не возможно, поскольку событие нажатия кнопки обрабатывается с помощью сообщений. Но поскольку данный пример использует TIdAntiFreeze, который вызывает Application.ProcessMessages и позволяет обрабатывать события отрисовки, так и другие события. По этой причине вы должны побеспокоиться о защите пользовательского интерфейса.

Используя TIdTCPClient (Client) – бросьте его на форму во время проектирования и попробуйте подключиться к серверу и подождите приветствия от сервера. GetResponse читает ответы и возвращает ответы как результат. В данном случае результат отбрасывается, но GetResult знает, что надо проверить ответ на число 204. Если Сервет отвечает другим кодом, то возбуждается исключение. Сервер может отвечать разными кодами, если он, например, очень, находится на профилактике и так далее.

Для каждого почтового индекса, который вводит пользователь, пример посылает команду lookup на сервер и ожидает код ответа 200. Если SendCmd закончится успешно, пример вызывает функцию Capture, которая читает ответы, пока не поступит с единственной точкой в строке. Поскольку демонстрационный пример за раз посылает одну команду, то ожидается одна строка в ответ, или ее отсутствие если индекс неверный.

По окончанию пример шлет команду Quit и ожидает ответа с кодом 201, который означает, что сервер понял и отсоединяет клиента. Правильным поведением, является всегда посылка команды Quit, чтобы обе стороны знали что произошло разъединение

7. UDP

7.1. Обзор

UDP (*User Datagram Protocol*) используется для датаграмм (*datagram)* и без установки соединения. UDP позволяет посылать короткие пакеты на узел, без установки соединения с ним. Для UDP пакетов не гарантируется доставка и не гарантируется тот же порядок приема, в каком они были посланы. При посылке UDP пакета, он посылается в одном блоке. Поэтому вы не должны превышать максимальный размер пакета, указанный в вашем TCP/IP стеке.

Из-за этих факторов, многие люди считают UDP малопригодными. Но это не совсем так. Многие потоковые протоколы, такие как Real Audio, используют UDP.

Примечание: термин "потоковый" может быть легко перепутан с термином поток (*stream*) для TCP. Когда вы видите эти термины, вы должны разобраться с контекстом, в котором они применяются, для определения точного значения.

7.2. Надежность

Надежность UDP пакетов зависит от надежности и загруженности сете. UDP пакеты, в основном используются в локальных сетях, поскольку локальные сети очень надежны. UDP проходящие через Интернет как правило также надежны и могут использоваться совместно с коррекцией ошибок и более часто с интерполяцией. Интерполяция основывается на воссоздании пропущенных данных на основе пакетов, принятых перед потерянным пакетом и/или перед, и после потерянного пакета. Доставка, конечно, не гарантируется в любой сети, поскольку предполагается, что данные всегда достигают получателя.

Поскольку UDP не имеет средств подтверждения доставки, то нет и гарантии ее доставки. Если вы посылаете UDP пакет на другой узел, вы не имеете возможности узнать доставлен ли пакет. Стек не в может и не будет определять это и не выдаст никакой ошибки если пакет не поступит получателю. Если вам требуется такое подтверждение, то вы должны сами разработать механизм подтверждения.

UDP аналогичен посылке сообщения на обычный пейджер. Вы знаете, что вы послали, но вы не знаете получено ли оно. Пейджер может не существовать, или находиться вне зоны приема, или может быть выключен, или не работоспособен, в дополнение сеть может потерять ваше сообщение. Пока обратная сторона не сообщит о приеме сообщения, вы этого не узнаете. В дополнение, если вы посылаете несколько сообщений, то возможно они поступят совсем в другом порядке.

Другой пример из реальной жизни – это отправка письма. Вы можете его послать, но не гарантируется, что оно будет доставлено получателю. Оно может быть потеряно где ни будь.

7.3. Широкополосносные сообщения (*Broadcast*)

UDP имеет уникальную возможность, что его делает очень применимым. Это возможность широкополосной посылки. Широкополосность (*Broadcast*) означает, что может быть послано одно сообщение, по получено многими получателями. Это не то же самое, как многоадресность (*multicasting*). Многоадресность – это модель подписки. Когда получатели делают подписку на получение и они добавляются в список рассылки. С помощью широкополосной рассылки, сообщение посылается по сети и все кто его слушают, могут принять его, без необходимости подписываться.

Многоадресные сообщения подобны газетной рассылки. Только те люди, которые подписались на доставку, получают его. Широкополосные сообщения аналогичны радиовещанию (От переводчика: в английском языке это одно и тоже слово **broadcasting**). Любой, кто имеет радиоприемник, может настроить его на любую радиостанцию и принимать ее. Радиослушатель не обязан оповещать радиостанцию, что он хочет слушать.

Конкретный IP адрес для сообщения, можно рассчитать, основываясь на IP отправителя и маски сети. Есть также особый случай, это адрес 255.255.255.255, который так далеко, насколько это возможно.

Но почти все маршрутизаторы по умолчанию настроены так, чтобы фильтровать широкополосные сообщения. Это означает, что такие сообщения не проходят через мосты или внешние маршрутизаторы и зона их распространения ограничена локальной сетью.

7.4. Размеры пакетов

Большинство операционных систем позволяют иметь размер UDP пакета 32K или даже 64K. Но, как правило, маршрутизаторы имеют более серьезные ограничения. UDP не должны превышать допустимый размер, разрешенный маршрутизатором или другим сетевым устройством. Но нет никакой возможности узнать это ограничение.

Поэтому рекомендуется, чтобы UDP пакеты были длиной в 8192 байт или менее, при передаче за пределами локальной сети. Но в некоторых случаях и этот размер велик. Для абсолютной гарантии делайте ваши пакеты в 1024 байт или менее.

7.5. Схемы подтверждений

7.5.1. Обзор

В локальной сети надежность UDP достаточно высокая. Но в случае WAN или Интернет вы, возможно, пожелаете разработать какую ни будь схему подтверждений.

7.5.2. Схема с подтверждениями

В случаи схемы подтверждения, каждый пакет должен быть подтвержден получателем, как принятый. Если подтверждение не получено в течение определенного периода, то пакет повторно отправляется получателю.

Поскольку подтверждения также могут быть потеряны, каждый пакет должен иметь уникальный идентификатор. Обычно идентификатор это просто последовательный номер. Это позволяет фильтровать повторно полученные пакеты и также посылать сообщения подтверждения, какие именно пакеты он получит.

7.5.3. Схема с последовательностями

Пакеты могут идентифицироваться последовательным номером. Этот номер может быть использован для определения потерянных пакетов. Это также может быть использовано, для запроса повторной передачи потерянного пакета, а для таких пакетов, как передача звука, использовать интерполировать данные на основе полученных и потерянных пакетов, или просто игнорировать потерянный пакет.

Такое поведение при потерянных пакетов можно услышать при передачи реальной речи при передачи по сотовому телефону. В некоторых случаях вы слышите пропуски или заикания.

7.6. Компонент TIdUDPClient

TIdUDPClient – это базовый компонент для посылки UDP пакетов. Наиболее часто используемый метод – это Send, в котором используются свойства Host и Port для посылки UDP пакета. Аргументом функции является строка.

Есть также метод SendBuffer, который выполняет туже задачу, но аргументами являются Buffer и Size.

TIdUDPClient также может использоваться как сервер, для приема входящих UDP пакетов.

7.7. Компонент TIdUDPServer

Поскольку UDP работает без соединений, то TIdUDPServer немного иначе, чем TIdTCPServer. TIdUDPServer не имеет режимов подобным TIdSimpleServer, Так как UDP не использует соединений, то TIdUDPClient (видимо здесь опечатка и имеется в виду TIdUDPServer) имеет только простые слушающие методы.

При активации TIdUDPServer создает слушающий поток для входящих UDP пакетов. Для каждого принятого UDP пакета, TIdUDPServer возбуждает событие OnUDPRead в главном кодовом потоке, или в контексте слушающего потока, в зависимости от свойства ThreadedEvent.

Когда свойство ThreadedEvent сброшено, то событие OnUDPRead возбуждается в контексте главного кодового потока. Когда свойство ThreadedEvent установлено, то событие OnUDPRead возбуждается в контексте слушающего потока.

Вне зависимость от состояния ThreadedEvent true или false, блокируется обработка других сообщений. Поэтому обработка OnUDPRead по возможности должна быть короткой.

7.8. UDP пример - RBSOD

7.8.1. Обзор

В данной главе приводится пример UDP клиента и UDP сервера. Данный пример это реальное приложение, которое может быть использовано для шуток в корпоративное среде. Тем не менее, он должен использоваться с осторожностью.

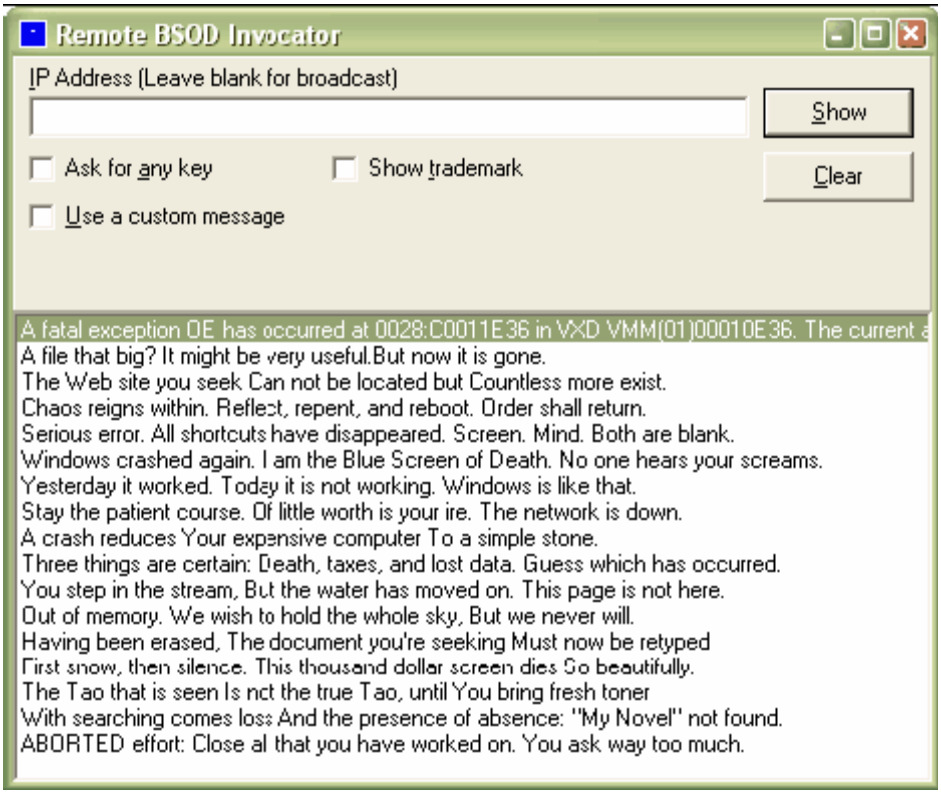
Пример называется «Remote BSOD invocator» - или сокращено RBSOD. RBSOD может быть использован для создания ложных BSOD (От переводчика: Blue Screen Of Dead – это знаменитый экран смерти) на машинах коллег (или врагов). BSOD не является реальным BSOD, и не делает ни каких потерь данных у жертвы. Тем не менее, он должен напугать его startle. Сам BSOD также может быть сброшен удаленно.

Примечание: по современной антивирусной терминологии, подобная программа считается вирусом- Joke Virus и очень опасно, поскольку с этим часто борются методом переустановки системы с нуля. Кроме того, это заготовка для более опасного трояна.

RBSOD состоит из двух программ. Сервер, который должен быть запущен на удаленной машине и клиент для управления сервером.

Предоставлено много параметров. В зависимости от выбранных параметров, BSOD может быть почти настоящим или появляться как шутка (ничего себе шуточки, особенно учитывая возможности бродкаста). Хотя это появляется как шутка, но клиенту нужно несколько секунд, чтобы понять это и течении этого периода он испытывает настоящий страх.

Клиентская часть RBSOD выглядит следующим образом:



Предоставлены следующие параметры.

7.8.1.1. IP Address

Укажите IP адрес или имя узла жертвы, у которой должен появиться, сервер конечно должен быть уже установлен и запущен на удаленном компьютере.

Если оставить это поле пустым, то сообщение будет послано в широковещательном режиме по всей подсети (или докуда оно сможет дойти) и на всех компьютерах, на которых установлен сервер возникнет BSOD.

7.8.1.2. Поле Message

Поле Message задает текст сообщения на машине жертвы, которое будет указано в BSOD screen. Значение по умолчанию следующее:

A fatal exception OE has occurred at 0028:C0011E36 in VXD VMM(01)00010E36. The current application will be terminated. (Произошла фатальная ошибка OE по адресу 0028:C0011E36 в VXD VMM(01)00010E36. Текущее приложение будет закрыто).

Данный текст взят из реального BSOD и будет казаться реальным для пользователя.

Множество японских хайку (особый вид японской поэзии) также включены для развлечения. Многие из них очень развлекательны, просто выберите одно из них. У меня есть два любимых:

Windows crashed again. I am the Blue Screen of Death. No one hears your screams. (Виндоус снова упал. Я синий экран смерти. Никто не услышит ваших криков).

Three things are certain: Death, taxes, and lost data. Guess which has occurred. (Три вещи неотвратимы: смерть, налоги и потерянные данные. Отгадайте, какая случилась).

Представляет вид ваших коллег, после того, как они поймут, что их разыграли. Но будьте особенно осторожны и не применяйте это к вашему шефу или к Бейсик программистам. Они особенно оценят вашу «шутку».

Сообщения находятся в файле messages.dat и вы можете добавить свои.

7.8.1.3. Use a Custom Message

Если данный параметр отмечен, то появляется другой диалог в котором вы можете ввести свое сообщение. Данный параметр полезен для организации интерактивных и более уместных BSOD сообщений. Например, если ваш шеф в необычной одежде, то вы можете создать такое сообщение:

Ugly brown suit error. I cannot continue in such company. (Очень неприятная ошибка в одежде. Я больше не могу оставаться далее в такой компании).

7.8.1.4. Show Any Key

Данный параметр, может быть использован для дополнительных выходок. BSOD выдает подсказку «нажмите любую клавишу для продолжения». Это так в нормальном BSOD. НО в данном случае, если будет отмечен, то после нажатия любой клавиши, будет выдано другое сообщение с мигающим текстом «Это не та клавиша, нажмите клавишу NY!».

Этот параметр должен отвратить вашего босса или Бейсик программиста потратить часы на поиск клавиши. Данный параметр наиболее хорошо использовать до отправки в аэропорт для длительного путешествия или когда вы хотите занять его надолго. (А себя в длительных поисках новой работы)

7.8.1.5. Show Trademark

Если данный параметр используется, то в нижнем углу экрана будет мелкая, но читаемая надпись: *** The BSOD is a trademark of the Microsoft Corporation.**

7.8.1.6. Клавиша Show

Нажатие клавиши **Show** приведет к генерации BSOD на удаленном компьютере.

7.8.1.7. Клавиша Clear

Нажатие клавиши Clear используется для удаленного снятия «BSOD». Обычно вы отдаете это на откуп пользователю, но в некоторых случая вы, возможно, захотите сами его снять. Ну, например, перед походом к пользователю.

7.8.2. Сервер RBSOD

Но сначала посмотрим на сервер.

7.8.2.1. Установка

Сервер назван svchost (еще одна любимая вещь у вирусов) вместо RBSODServer или другим более говорящим именем. Это сделано, чтобы для более простой установки на другом компьютере и одновременно скрываете его. Имя svchost – это нормальное имя системного исполнимого файла, который запускает множество копий исполнимых файлов. Если вы посмотрите в диспетчере задач, то вы несколько запущенных копий подобного процесса. Поскольку вы разместите свой специальную копию в другой папке, а не в системной, то он не будет драться с родным, но будет выглядеть как нормальный в диспетчере задач.

Сервер не имеет окон и не появляется в панели задач и системном трее (там, где часики). Если вы хотите его остановить, то запустите диспетчер задач и выберите тот svchost, который запущен от имени пользователя. Системный svchosts запущен от SYSTEM (это не совсем так). После выбора вашей «версии» нажмите клавишу [**Снять задачу**].

Для инсталляции просто скопируйте сервер на машину клиента (compile without packages for easiest deployment) и запустит его. Он останется в памяти пока пользователь не перезагрузит свой компьютере. После перезагрузки программа не будет перезапущена. Если вы желаете, что бы программа автоматически стартовала при каждой перезагрузки, то просто добавьте ее в автозагрузку или в реестр, для невидимого запуска.

7.8.2.2. Исходный код

Сервер состоит из двух частей, Server.pas и BSOD.pas. BSOD.pas содержит форму, которая используется для показа BSOD экрана. BSOD.pas не содержит Indy кода и поэтому не рассматривается здесь.

Server.pas – это главная форма приложения и содержит один UDP сервер. Свойство port установлено в 6001 и свойство active установлено в True. Когда приложение запускается, то оно немедленно начинает слушать порт для входящих UDP пакетов.

Как ранее обсуждалось UDP подобен пейджеру. Поскольку ему не требуется соединение для приема данных. Пакеты данных просто появляются как один кусок. Для каждого принятого UDP пакета, UDP сервер возбуждает событие OnUDPRead. Больше никаких других событий не требуется для реализации UDP сервера. Когда возбуждается событие OnUDPRead, полный пакет уже принят и готов к использованию.

Событию OnUDPRead передаются три аргумента:

1. ASender: TObject – это компонент, который возбудил данное событие. Это применимо только если будет создано несколько UDPServers серверов и используют один и тот же метод для события. Это очень редко нужно.
2. AData: TStream – это основной аргумент и он содержит сам пакет. UDP пакеты могут содержать текст и/или двоичные данные. Поэтому Indy предоставляет их в виде потока. Для доступа к данным, просто используйте методы read класса TStream.
3. ABinding: TIdSocketHandle – этот аргумент пригоден для получения расширенной информации. Это востребовано, если используется связывание (*bindings*). Вот пример как выглядит событие OnUDPRead в RBSOD сервере:

```
procedure TformMain.IdUDPServer1UDPRead(Sender: TObject; AData: TStream;
  ABinding: TIdSocketHandle);
var
  LMsg: string;
begin
  if AData.Size = 0 then
  begin
    formBSOD.Hide;
  end
  else
  begin
    // Move from stream into a string
    SetLength(LMsg, AData.Size);
    AData.ReadBuffer(LMsg[1], Length(LMsg));
    //
```

```
      formBSOD.ShowBSOD(Copy(LMsg, 3, MaxInt),
         Copy(LMsg, 1, 1) = 'T',
         Copy(LMsg, 2, 1) = 'T');
    end;
  end;
```

Обратим внимание, что оператор if проверяет на нулевую длину. Это вполне легально посылать и принимать пустые UDP пакеты. В данном случае это используется для сигнализации серверу о снятии BSOD.

Если размер не равен 0, то данные копируются в локальную строковую перемену с помощью TStream.ReadBuffer.

UDP сервер не использует отдельного потока для каждого пакета, поскольку события OnUDPRead возникают последовательно. По умолчанию OnUDPRead возникает в главном кодовом потоке и формы и другие GUI органы управления могут использоваться безопасно.

7.8.3. Клиент RBSOD

RBSOD клиент еще проще чем сервер. RBSOD клиент состоит из одной формы: Main.pas. Main.pas содержит несколько событий, но большинство из них относятся к пользовательскому интерфейсу и понятны сами по себе.

Основной Indy код в клиенте RBSOD, который находится в обработчике the OnClick клавиши Show.

```
IdUDPClient1.Host := editHost.Text;
IdUDPClient1.Send(
  iif(chckShowAnyKey.Checked, 'T', 'F')
  + iif(chckTrademark.Checked, 'T', 'F')
  + s);
```

Первая строка устанавливает узел, на который будет посылаться UDP пакет. Порт уже установлен в значение 6001, с помощью инспектора объектов.

Следующая строка использует метод Send для посылки UDP пакета. Поскольку UDP не требует соединения, любые данные могут быть посланные как несколько пакетов или быть собраны в один пакет. Если посылается несколько пакетов, то разработчики должны обеспечить их сборку, координацию и разборку. Это не совсем тривиальная задача, то проще собрать большое количество данных в один пакет и послать его.

Аргумент, переданный для пересылки, немедленно отсылается как UDP пакет и поэтому все данные для пересылки должны быть посланы с помощью одного пакета.

Indy также содержит перегруженный метод SendBuffer для посылки данных из буферов.

В случае RBSOD просто содержит два символа, которые определяют, как показывать торговую марку и показывать any key, затем актуально сообщение для отображения.

Еще есть другой Indy код в клиенте RBSOD, который находится в обработчике события OnClick для клавиши Clear и он почти идентичен приведенному ранее отрывку.

8. Чтение и запись

Indy поддерживает несколько методов, для чтения и записи, которые отвечают различным потребностям. Эти методы также включают – ожидание, проверку состояния и опрос.

Каждый из методов – это подмножество класса TIdTCPConnection. Это зависит от типа каждого серверного соединения и являются наследником TIdTCPClient. Это означает, что вы можете использовать эти методы, как для серверов, так и для клиентов.

Большинство людей знакомы только с некоторыми методами чтения и записи. Поскольку, многие люди никогда не используют ядро клиентов, а работают только с протоколами клиентов и не знакомы с методами ядра.

Вспомни, что Indy блокирующий, и не использует событий для оповещения об окончании запрошенной операции. Возврат из методов не происходит до окончания запрошенной операции. Если операция не может быть выполнена, то возбуждается исключение.

В данной книге не приводится полный набор документации по всем методам. Для этого вы должны смотреть справки по Indy. Данная глава просто написана для ознакомления с методами.

8.1. Методы чтения

8.1.1. Функция AllData

```
function AllData: string;
```

Функция AllData блокирует и собирает все входящие данные, до разъединения соединения. Затем возвращает все собранные данные в качестве результата. AllData полезен для протоколов аналогичным WhoIs, после получения запроса, возвращает данные и сигнал, что соединение прекращено. AllData хранит свои буферы в оперативной памяти, поэтому не стоит использовать ее для больших по размеру данных.

8.1.2. Процедура Capture

```
procedure Capture(ADest: TStream; const ADelim: string = '.';
const AIsRFCMessage: Boolean = True); overload;

procedure Capture(ADest: TStream; out VLineCount: Integer; const
ADelim: string = '.'; const AIsRFCMessage: Boolean = True);
overload;

procedure Capture(ADest: TStrings; const ADelim: string = '.';
const AIsRFCMessage: Boolean = True); overload;

procedure Capture(ADest: TStrings; out VLineCount: Integer; const
ADelim: string = '.'; const AIsRFCMessage: Boolean = True);
overload;
```

Процедура Capture существует в нескольких перекрытых формах. Суммируя, Capture читает данные, пока не встретит указанный ограничитель в строке.

8.1.3. Функция CurrentReadBuffer

```
function CurrentReadBuffer: string;
```

Функция CurrentReadBuffer возвращает все данные, которые находятся во внутреннем буфере Indy. Перед возвратом данных, CurrentReadBuffer также пытается прочитать все данные,

которые она ожидает от подсоединенного сокета. Вызов CurrentReadBuffer очищает внутренний буфер.

Если данных нет, то возвращается пустая строка.

8.1.4. Свойство InputBuffer

```
property InputBuffer: TIdManagedBuffer read FInputBuffer;
```

Свойство InputBuffer – это ссылка на экземпляр объекта TIdManagedBuffer. InputBuffer – это внутренний буфер Indy. TIdManagedBuffer имеет несколько расширенных буферов, но обычно пользователь не использует их.

8.1.5. Функция InputLn

```
function InputLn(const AMask: String = ''; AEcho: Boolean = True; ATabWidth:
Integer = 8; AMaxLineLength: Integer = -1): String;
```

Функция InputLn читает строку от сервере и возвращает е обратно reads a line from the server and echoes it back honoring the backspace character. Если параметр AMask указан, то строка AMask отсылается вместо каждого принятого символ. Функция InputLn полезна, когда вы не желаете отображать принятые символы, например пароль.

8.1.6. Процедура ReadBuffer

```
procedure ReadBuffer(var ABuffer; const AByteCount: Longint);
```

Процедура ReadBuffer используется для чтения данных напрямую в указанный буфер. Если буфер недостаточен, то процедура ReadBuffer читает данные во внутренний буфер.

8.1.7. Функция ReadCardinal

```
function ReadCardinal(const AConvert: boolean = true): Cardinal;
```

Функция ReadCardinal читает 32-битное число без знака из соединения, с дополнительным учетом сетевого порядка байт.

8.1.8. Функция ReadFromStack

```
function ReadFromStack(const ARaiseExceptionIfDisconnected: Boolean = True;
ATimeout: Integer = IdTimeoutDefault;
```

Функция ReadFromStack используется для заполнения внутреннего буфера Indy. Обычно конечные пользователи никогда не должны использовать эту функцию, поскольку она реализует новый метод чтения без использования текущих методов чтения, или когда они работают напрямую с внутренним буфером с помощью свойства InternalBuffer.

8.1.9. Функция ReadInteger

```
function ReadInteger(const AConvert: boolean = true): Integer;
```

Функция ReadInteger читает r 32-битное число со знаком из соединения, с дополнительным учетом сетевого порядка байт.

8.1.10. Функция ReadLn

```
function ReadLn(ATerminator: string = LF; const ATimeout: Integer =
IdTimeoutDefault; AMaxLineLength: Integer = -1): string; virtual;
```

Функция ReadLn читает данные из соединения, пока не встретит указанный ограничитель, в течение периода таймаута или в случае приема указанной максимальной длины строки.

8.1.11. Функция ReadLnWait

```
function ReadLnWait(AFailCount: Integer = MaxInt): string;
```

Функция ReadLnWait подобна функции ReadLn с одним исключением, что она не возвращает результат, пока не примет, не пустую строку. Она также вернет количество принятых пустых строк.

8.1.12. Функция ReadSmallInt

```
function ReadSmallInt(const AConvert: Boolean = true): SmallInt;
```

Функция ReadSmallInt читает короткое целое из соединения, дополнительно делая подстройку под сетевой порядок байт.

8.1.13. Процедура ReadStream

```
procedure ReadStream(AStream: TStream; AByteCount: LongInt = -1; const
AReadUntilDisconnect: boolean = false);
```

Функция ReadStream читает данные из потока. В функции может быть указано количество байт, для чтения из потока или до отсоединения.

8.1.14. Функция ReadString

```
function ReadString(const ABytes: Integer): string;
```

Функция ReadString читаетуказанное количество байт в строку и возвращает данные как результат.

8.1.15. Процедура ReadStrings

```
procedure ReadStrings(var AValue: TStrings; AReadLinesCount: Integer = -1);
```

Процедура ReadStrings читает указанное количество строк, разделенных символом EOLs из соединения. Если количество строк не указано, то читается первое 32-битное целое из соединение и оно используется далее как количество строк.

8.1.16. Функция WaitFor

```
function WaitFor(const AString: string): string;
```

Функция WaitFor читает данные из соединения, пока не встретит указанную строку.

8.2. Таймауты чтения

TIdTCPConnection (Все TCP клиенты и соединения, которые унаследованы от TIdTCPConnection) имеет свойство, названое ReadTimeout. Свойство ReadTimeout указывает таймаут в миллисекундах. Значением свойства по умолчанию является IdTimeoutInfinite. Данная установка запрещает таймауты.

Таймаут не является таймаутом окончания работы. Это просто пустой таймаут. Что это значит, если указанная величина в свойстве ReadTimeout прошло и не данных для записи, то будет возбуждено исключение EIdReadTimeout.

Многие сетевые соединения бывают очень медленные и данные не передавались, но соединение остается действительным. В такой ситуации, соединение может стать медленным и неприемлемым и будет только приводить к нагрузке сервера и малопригодным для клиента.

Чтобы можно было оперировать такой ситуацией Indy реализует таймауты с помощью свойство ReadTimeout класса TIdTCPConnection. Свойство ReadTimeout по умолчанию равно нулю, которое запрещает обработку таймаутов. Для разрешения обработки таймаутов установите его значение в миллисекундах.

Во время чтения, если в течение указанного интервала не будет приема данных из соединения, то возникнет исключение EIdReadTimeout. Таймаут не применяется если принимаются данные в течение периода. Если вы запросили 100 байт и таймаут 1000 миллисекунд (1 секунда) операция чтения может занять более одной секунды. От переводчика: если в будет в течение каждой секунды принимать по одному байту, то в результате прием займет 100 секунд.

Только если в течение одной секунды не будет принято ни одного байта, только тогда возникнет исключение EIdReadTimeout.

8.3. Методы записи

8.3.1. Функция SendCmd

```
function SendCmd(const AOut: string; const AResponse: SmallInt = -1): SmallInt;
overload;

function SendCmd(const AOut: string; const AResponse: Array of SmallInt):
SmallInt; overload;
```

Функция SendCmd используется для передачи текстовой команды и ожидает ответа в формате цифровых ответов RFC.

8.3.2. Процедура Write

```
procedure Write(AOut: string);
```

Процедура Write это наиболее общий метод вывода в Indy. Процедура Write посылает AOut в соединение. Процедура Write не изменяет AOut ни каким образом.

8.3.3. Процедура WriteBuffer

```
procedure WriteBuffer(const ABuffer; AByteCount: Longint; const AWriteNow:
Boolean = False);
```

Процедура WriteBuffer позволяет послать содержимое буфера. Если AWriteNow указано, то буферизация в процедуре write будет опущено, если оно в данный момент используется.

8.3.4. Процедура WriteCardinal

```
procedure WriteCardinal(AValue: Cardinal; const AConvert: Boolean = True);
```

Процедура WriteCardinal посылает 32-битное, целое без знака в соединение, дополнительно преобразовывая в сетевой порядок байт.

8.3.5. Процедура WriteHeader

```
procedure WriteHeader(AHeader: TStrings);
```

Процедура WriteHeader посылает объект TStrings в соединения, преобразовывая '=' в ': ' последовательность каждое вхождение item. Процедура WriteHeader также записывает пустую строку после передачи всего объекта TStrings.

8.3.6. Процедура WriteInteger

```
procedure WriteInteger(AValue: Integer; const AConvert: Boolean = True);
```

Процедура WriteInteger посылает 32-битное, целое знаковое в соединение, дополнительно преобразовывая в сетевой порядок байт.

8.3.7. Процедура WriteLn

```
procedure WriteLn(const AOut: string = '');
```

Процедура WriteLn выполняет те же функции, как и процедура Write с одним исключением, что она добавляет последовательность EOL (CR + LF) после AOut параметра.

8.3.8. Процедура WriteRFCReply

```
procedure WriteRFCReply(AReply: TIdRFCReply);
```

Процедура WriteRFCReply посылает цифру ответа + текст в RFC стиле, используя указанный TIdRFCReply.

8.3.9. Процедура WriteRFCStrings

```
procedure WriteRFCStrings(AStrings: TStrings);
```

Процедура WriteRFCStrings посылает TStrings ответ в стиле RFC сообщений, заканчивая строкой с '.' в начале.

8.3.10. Процедура WriteSmallInt

```
procedure WriteSmallInt(AValue: SmallInt; const AConvert: Boolean = True);
```

Процедура WriteSmallInt посылает small integer в соединение, дополнительно преобразовывая в сетевой порядок байт.

8.3.11. Процедура WriteStream

```
procedure WriteStream(AStream: TStream; const AAll: Boolean = True; const
AWriteByteCount: Boolean = False; const ASize: Integer = 0);
```

Процедура WriteStream посылает указанный поток данных (*stream*) в соединение. Процедура WriteStream содержит много параметров для указания какие части потока должны быть посланы и дополнительно может посылать количество байт в поток.

8.3.12. Процедура WriteStrings

```
procedure WriteStrings(AValue: TStrings; const AWriteLinesCount: Boolean =
False);
```

Процедура WriteStrings посылает объект TStrings в соединение и копию в ReadStrings.

8.3.13. Функция WriteFile

```
function WriteFile(AFile: String; const AEnableTransferFile: Boolean = False):
Cardinal;
```

Функция WriteFile – это функция прямой посылки содержимого файла в соединение. Функция WriteFile использует операционную систему, просто используя TFileStream вместе с SendStream.

8.4. Буферизация записи

TCP должен посылать данные пакетами. Размер пакетов может изменяться, но как правило он немногим менее 1 килобайта. Тем не менее, если TCP ожидает полного пакета данных, во многих случаях ответы могут быть не получены, поскольку запросы не были отосланы. TCP может использовать полные или неполные пакеты, Для неполных пакетов можно использовать алгоритм названый Nagle Coalescing. Nagle Coalescing внутренне буферизует данные, пока не будет достигнут указанный размер пакета или не истечет рассчитанное количество времени. Период времени обычно маленький и измеряется в миллисекундах.

Посылка множества маленьких кусков данных может сделать алгоритм неэффективным, поскольку каждый пакет имеет служебный заголовок, что перегружает полосу пропускания и уменьшает скорость передачи данных.

Данные могут буферизироваться в строку и быть посланы за раз, тем не менее это требует писать дополнительный код и исключает использование всех методов записи во время буферизации. Это также может усложнить ваш код и увеличить потребности в памяти.

Вместо этого, вы можете использовать возможности Indy по буферизации записи. При использовании методов записи с буферизацией, вы можете позволить Indy буферизировать исходящие данные, что позволит вам использовать все возможные методы записи.

Для начала буферизации записи, вызовите метод OpenWriteBuffer и укажите размер буфера. Затем вы можете вызывать все функции записи в Indy, и весь вывод будет буферизироваться, пока буфер не будет заполнен. Каждый раз когда будет достигнут размер буфера, буфер посылается в соединение и очищается. Если размер буфера не указан, все данные буферизуются и должны быть вручную отправлены.

Имеется также несколько методов для управления буфером.

ClearWriteBuffer очищает текущий буфер и сохраняет буфер открытым. FlushWriteBuffer сбрасывает текущее содержимое и также сохраняет буфер открытым.

Для прекращения буферизации записи, вызовите CancelWriteBuffer или CloseWriteBuffer. CloseWriteBuffer записывает имеющие данные и заканчивает буферизацию записи, а CancelWriteBuffer закрывает буферизацию записи, без передачи.

8.5. Работа транзакций

Транзакции используются для определения единиц работы. Они называются рабочие транзакции и могут быть вложенными, и отделяют операции чтения и записи, которые могут быть одновременными. Рабочие транзакции обычно используются для отображения прогресса и состояния передачи.

Транзакции определены в Indy различными, предопределенными методами, такими как TIdHTTP.Get, WriteStream и так далее. Как пользователь, вы также можете определить свои собственные транзакции, с помощью BeginWork, DoWork и EndWork.

8.5.1. События OnWork

События OnWork состоят из трех событий и используется для связи состояний транзакций. Эти три события следующие: OnWorkBegin, OnWork и OnWorkEnd.

При начале транзакции возникает событие OnWorkBegin. В событие OnWorkBegin передаются Sender, WorkMode и WorkCount. Параметр Sender это соединение, к которому относится транзакция. Параметр WorkMode указывает режим работы – читающая или пишущая транзакция. Транзакции Read и Write могут возникать одновременно и транзакции могут быть вложенными. Параметр WorkCount указывает размер транзакции. Во многих транзакциях, размер не может быть определен, и в этом случае, WorkCount имеет значение 0. Иначе, WorkCount показывает количество байт. Обычно данное событие используется для отображения начала процесса.

Затем возникает серия событий OnWork. В событие OnWork передаются Sender, WorkMode и текущий WorkCount. Данное событие используется для отображения прогресса.

Когда транзакция заканчивает, возникает событие OnWorkEnd. В событие OnWorkEnd OnWork передаются только Sender и WorkMode. Данное событие используется для отображения завергения процесса.

8.5.2. Управление своими собственными рабочими транзакциями

Вы можете создавать свои собственные транзакции, вызывая BeginWork, DoWork и EndWork. Параметры те же самые, что описанны выше. Вызовы вложенных транзакций обслуживаются автоматически.

Для выполнения транзакции, сначала вызовите BeginWork с указанием размера, если он известен. Затем вызовайте DoWork для отображения прогресса. По окончанию вызовите EndWork.

9. Обнаружение разъединения

Поскольку Indy блокирующая по природе, и ее события используются только для обработки состояния, поэтому в ней нет событий оповещения о неожиданном разъединении соединения. Если вызов чтения или записи находится в состоянии исполнения и происходит неожиданный разрыв соединения, то вырабатывается исключение. Если же в этот момент нет вызова на чтение/запись, то исключение возникнет при первом вызове.

Имеется событие OnDisconnected, но это не то, о чем вы подумали. Событие OnDisconnected возникает при вызове метода Disconnect. Это событие никак не связано с неожиданным разъединением.

9.1. Скажем прощай

В TCP протоколах, базирующихся на командах, самый простой путь завершить работу это сказать прощай (good bye). Эта команда распознается, только при наличии действующего соединении, но даже это большое удобство.

Чтобы завершить работу, протоколы используют команду QUIT. Когда клиент готов завершить работу, вместо немедленного разъединения, сначала попробуйте послать команду QUIT на сервер. Затем подождите ответа и только после этого производите отсоединение сокета.

Это хорошие манеры и позволяет обеим сторонам произвести правильное разъединение. Не делайте это как в телефонном разговоре, просто вешая трубку. Вы же не знаете, что произойдет в данном случае.

Разница в том, что сервер может иметь сотни и тысячи клиентских соединений. Если они все отсоединятся без оповещения сервера, то сервер будет продолжать держать кучу мертвых соединений.

9.2. А нужно ли вам реально знать это?

Многие программисты ошибаются, считая, что они должны немедленно узнать, что соединение неожиданно прервалось. Вы наверно слышали следующую сентенцию – «если дерево падает в лесу и никто при этом не присутствует, то кто услышит звук?». Если сокет отсоединяется, и он становится не доступным, то в действительно ли сокет закрывается? В большинстве случаев ответ НЕТ. Если сокет физически отсоединен и недоступен, то исключение не будет возбуждено, до тех пор, пока не будет произведена попытка доступа к сокету.

Это может казаться странным, но это также как при работе с файлами. Представим себе, что вы открыли таблицу Экселя с гибкого диска, и затем вынули диск из дисковода. Вы можете продолжать работать с файлом, потому что он кэширован в памяти и Эксель, в данный момент, не обращается к физическому файлу. Иногда в течение этого времени вы забываете, что вы вынули дискету из дисководу. Позже вы продолжаете работу с вашей таблицей, все прекрасно пока вы не захотите сохранить ваши изменения. Только в этот момент вы получите сообщение об ошибке. Эксель не получает "EAnIdiotRemovedTheFloppyDiskException", когда вы вынули дискету, но он имеет открытый файл на этом диске.

9.3. Я должен знать это немедленно!

Повторим, что исключение не возникает немедленно. Например, если вы вытащите сетевой кабель из платы или коммутатора, то может потребоваться минута или более. TCP/IP был разработан военными Америки, чтобы иметь устойчивый сетевой протокол, способный противостоять ядерным атакам. Поэтому он спроектирован так, что сеть ждет некоторое время

ответов с другой стороны соединения и пытается делать повторные запросы. Вы можете детектировать разрыв соединения немедленно, но иным путем, чем это реализовано в TCP/IP. Данное детектирование может быть реализовано, но очень важно понять, почему для этого нет стандартных функций. Это понимание поможет вам понять, как это можно реализовать.

В большинстве случаев, если вы думаете об этом, вы поймете, что данное поведение приемлемое и даже желательное.

Тем не менее, есть ситуации, когда очень важно немедленно знать о потере соединения. Вообразим, что вы реализовали интерфейс между монитором сердечной деятельности и телеметрической системой. В этом случае, вы определенно желаете знать, что соединение потеряно как можно раньше.

Если вы нуждаетесь в немедленном оповещении о разъединении, то вы должны встроить эту возможность в протокол.

9.3.1. Keep Alives

Keep alive (*хранить живым*) – это один из методов, детектирования немедленных разъединений. Keep alive реализован так, что одна сторона соединения на основе регулярного интервала посылается сообщение, такое как NOOP (*No Operation – нет операции*) другой стороне и всегда ожидает ответа. Если ответ не получен в течение указанного времени, то соединение считается проблематичным и уничтожается. Период таймаута достаточно короткий и определятся задачами протокола. Если вы реализуете соединение с сердечным монитором, то вы надеетесь, что указанный таймаут короче, чем таймаут для контроля температуры бочонка с пивом.

Есть и другие преимущества при реализации системы keep alive. Во многих случаях, соединение TCP может оставаться действительным, даже если процесс перестал отвечать и принимать сообщения. В других случаях, процесс может продолжать обслуживать запросы, но скорость может быть очень низкой из-за проблем в сервисе, или из-за уменьшения полосы пропускания сети. Keep alive может определять эти ситуации и отмечать их как непригодные, хотя соединение еще существует.

Keep alive запрос может посылаться на регулярной основе или при необходимости, когда требует определить состояние, в зависимости от требований протокола.

9.3.2. Пинги (*Pings*)

Пинги – это реализация, подобная keep alive, за исключением, что они не возвращаются в ответе на запрос. Пинги посылаются с одной стороны соединения, на другую на основе регулярного интервала. Одна сторона становится отправителем, а вторая монитором. Если в течение указанного интервала времени на мониторе, ответа не будет, то соединение считается непригодным и уничтожается.

Пинги могут считаться аналогичными биениям сердца. Если они становятся редкими, то это означает проблему.

9.4. Исключение EIdConnClosedGracefully

Многих Indy пользователей беспокоит исключение EIdConnClosedGracefully, которое часто возбуждается серверами, особенно HTTP и другими серверами. Исключение EIdConnClosedGracefully сигнализирует, что другая сторона умышленно закрыла соединение. Это не тоже самое, как потерянное соединение, которое может появляется в случае ошибки соединения. Если другая сторона закрыла соединение и сокет пытается писать или читать из

него, то возбуждается исключениеEIdConnClosedGracefully. Это подобно попытке чтения или записи в файл, который был закрыт без вашего оповещения.

В некоторых случаях – это подлинное исключение и ваш код должен его обработать. В других случаях (обычно в серверах) это нормальное функционирование протокола и Indy обработает это за вас. Даже если Indy поймала его, когда вы работали в отладчике, то оно возбуждается в нем. Вы просто должны нажать F9 и продолжать и Indy обслужить это исключение, но отладчик постоянно надоедает вам. В том случае когда Indy ловит подобное исключение, ваши пользователи никогда не видят его в ваших программах, если только программа не запущена под отладчиком IDE.

9.4.1. Введение

9.4.2. Почему случаются исключения на серверах?

Когда клиент подсоединяется к серверу, то имеются два пути к разъединению соединения:

1. Взаимное соглашение – обе стороны согласны взаимно разъединиться, если одна из сторон попросит об этом и обе стороны явно разъединяются.
2. Одностороннее разъединение – разъединение и оповещение другой стороны об этом.

С помощью взаимного соглашения обе стороны знают когда разъединяться и обе явно производят разъединение. Большинство протоколов обмена, такие как Mail, News и другие разъединяются подобным образом.

Когда клиент готов разъединиться, то он посылает команду серверу, оповещая его о своем желании. Сервер отвечает подтверждением и затем обе стороны закрывают соединение. В этих случаях исключение EIdConnClosedGracefully не возникает и если это случается, то это означает ошибку, которая должна быть соответственно обработана.

В некоторых случаях взаимного соглашения не используется команда, но обе стороны знают когда другая может отсоединиться. Часто клиент подсоединяется, посылает одну команду, получает ответ и отсоединяется. Когда не используются явные команды для отсоединения, протокол определяет, что клиент должен отключиться от сервера после посылки команды и получения ответа. Примером этого являются некоторые протоколы времени.

При одностороннем разъединении одна сторона просто отсоединяется. Другая сторона продолжает следить и как только определяет разрыв, то прекращает сессию. В протоколах, которые используют это, вы получите исключение EIdConnClosedGracefully и это нормальное поведение для них. Это исключение, но Indy в курсе и обработает это за вас. Протокол whois пример данного поведения. Клиент подсоединяется к серверу и посылает строку запроса. Сервер затем посылает ответ и отсоединяется. Ни каких других сигналов клиенту не посылается, просто производится разъединение.

Протокол HTTP позволяет оба вида – разъединение по взаимному соглашению и одностороннее разъединение и поэтому понятно, почему вы видите иногда исключение EIdConnClosedGracefully с HTTP серверами. Протокол HTTP 1.0 работает подобно протоколу whois, в том смысле, что сервер не посылает сигнала о разъединении и клиент должен просто разъединиться после получения ответа. Клиент должен использовать новое соединение для каждого запроса.

Протокол HTTP 1.1 позволяет в одном соединение запрашивать несколько документов. Тем не менее нет команды разъединения. В любое время, или клиент, или сервер могут разъединиться после получения ответов. Когда клиент разъединяется, но сервер еще продолжает воспринимать

запросы, то возникает исключение EIdConnClosedGracefully. В большинстве случаев в HTTP 1.1, клиент единственный, кто разъединяется. А сервер разъединяется как сделано в части реализации протокола HTTP 1.1, если не используется установка keep alive.

9.4.3. Почему это исключение?

Многие пользователи комментируют, что здесь должен быть сигнал об отсоединении, а не исключение. Но это неверный подход в данном случае.

Исключение EIdConnClosedGracefully возбуждается из ядра, глубоко внутри, в нескольких методах. Исключение EIdConnClosedGracefully в действительности исключение и в большинстве случаев должно быть поймано самым верхним уровнем для обработки. Исключение - это правильный путь для этого.

9.4.4. Это ошибка?

Не все исключения – это ошибки. Многие разработчики считают и уверены, что все исключения это ошибки. Это не так. Именно поэтому они названы исключениями, а не ошибками.

Delphi и C++ Builder используют исключения, для обработки ошибок, элегантным путем. Тем не менее исключения могут использоваться не только в случае ошибок. Примером этого является EAbort. Исключения подобного рода используются для изменения потока исполнения и для связи с более высоким уровнем в программе, где они будут пойманы. Indy использует исключения подобным образом.

9.4.5. А когда это ошибка?

Когда исключение EIdConnClosedGracefully возникает на клиенте, то это является ошибкой и должно быть поймано и обработано.

На сервере это исключение. Тем не менее иногда это ошибка и иногда это исключение. Во многих протоколах данное исключение это часть нормального функционирования протокола. Это общее поведение, и если вы не ловите EIdConnClosedGracefully на вашем сервере, то Indy сделает это. Это пометит соединение как закрытое и остановит рабочий поток, назначенный соединению. Если вы желаете, вы можете сами обработать данное исключение, иначе Indy сделает это за вас.

9.4.6. Простое решение

Исключение EIdConnClosedGracefully это базовый класс исключения, особенно для некоторых серверов, оно унаследовано от EIdSilentException. На закладке Language Exceptions tab of Debugger Options (Tools Menu) вы может добавить исключение EIdSilentException в список игнорируемых исключении. После этого добавленное исключение, если оно случится в коде будет обработано в программе, но отладчик не будет прерывать программу для обработки в среде.

10. Реализация протоколов

В Indy реализовано большинство распространенных протоколов. Но все равно бывают случаи, когда требуется реализовать протокол самостоятельно. Основная причина в необходимости реализации состоит в том, что нужный протокол отсутствует.

Первый шаг в понимание что же надо реализовать. Имеется три базовых типа протоколов:

1. **Стандартный** – эти протоколы относятся к Интернет стандартам. Для понимания просто найдите в RFC нужный протокол. В RFC протокол рассмотрен в подробностях.
2. **Пользовательский** – пользовательские протоколы используются, когда отсутствует нужный протокол. Создание пользовательского протокола будет описано позже.
3. **Разработчика** – это протоколы, предназначенные для общения с собственными системами или оборудованием. Протоколы разработчика – это пользовательский протокол, разработанный программистом, а вы общаетесь с ним. Я желаю вам удачи, поскольку большинство протоколов разработчика реализованы специалистами по оборудованию, которые имели один курс обучения в колледже и не имеют опыта разработки протоколов.

Большинство протоколов общаются, с помощью текста, и если только у вас нет особых причин, то и вы тоже должны поступать также.

Первым шагом построения клиента или сервера – это понять протокол. Для стандартных протоколов это может быть сделано с помощью чтения RFC. Для пользовательских протоколов вы его должны сами разработать.

Большинство протоколов используют просто текст. Общение означает посылку команд и получение ответов, а возможно и данных. Текст делает протокол более простым для отладки и позволяется использовать любые языки программирования и операционные системы.

Код состояния традиционно трехзначный номер. Не имеется стандарта на определение кодов состояния, но многие протоколы используют де-факто следующие соглашения:

- 1xx - информационные
- 2xx - успешные
- 3xx – временные ошибки
- 4xx – постоянные ошибки
- 5xx – внутренние ошибки

Каждой ошибке обычно назначается уникальный номер, но некоторые протоколы используют номера повторно и только предоставляют уникальный цифровой номер для каждой команды, а не протокола.

10.1. Терминология протокола

Перед обсуждением реализации протокола рассмотрим некоторые термины.

10.1.1. Простой текст (*plain text*)

Простой текст (*plain text*) означает, что все команды (*commands*) и ответы (*replies*) используют только 7-битный код ASCII. Если передача данных в откликах (*responses*) допустима в двоичном виде, то почти все команды протокола и ответы используют простой текст. Двоичный

код никогда не должен использоваться в командах и ответах, если на это не будет веских оснований.

Использование простого текста упрощает отладку и тестирование. Это также означает переносимость между операционными системами и языками программирования.

10.1.2. Команды (*commands*)

Команда это текстовая строка, которая посылается клиентом серверу для запроса информации или выполнения определенных действий. Примеры команд: HELP, QUIT, LIST.

Команды могут содержать дополнительные параметры, разделенные пробелами.

Пример:

```
GET CHARTS.DAT
```

GET - это команда, а CHARTS.DAT – это параметр. В данном примере используется только один параметр, но команды могут содержать несколько параметров. Обычно параметры разделены пробелами, аналогично параметрам в DOS или в командной строке.

Команды всегда кодируются на английском языке. Это может казаться предвзятым, тем не менее это общая практика, как в техническом мире, так и в мире бизнеса. Если английский не используется, то протокол не очень применим.

Один из плохих примеров последствий локализации команд, является Microsoft Office. Microsoft Office может быть автоматизирован, с помощью OLE. Тем не менее, Microsoft локализовал имена методов и свойств объектов. Это означает, что если вы пишите приложения с помощью Microsoft Office Automation в американской версии, то ваше приложение не будет работать во французской версии.

10.1.3. Ответы (*reply*)

Ответы – это короткий ответ на посланную команду. Ответ содержит информацию о состоянии – успешно ли ошибка, и иногда содержит небольшое количество данных.

Например, если команда *GET customer.dat*, вернет ответ *200 OK,* это будет означать, что команда воспринята и будет обработана. Ответы обычно состоят только из одной строки, но могут состоять и из нескольких строк.

В отличие от команд, текстовая часть ответа может быть локализована на другой язык, если она соответствует ограничению 7-бит ASCII. Поскольку протокол использует цифровую часть, а текстовая часть используется только для конечного пользователя и для отладки.

10.1.4. Отклики (*response*)

Отклик – это часть данных, которая возвращается в ответ на команду. Отклики дополнительны и не присутствуют во всех команда. Отклики посылаются после ответа, для распознавание правильный ли ответ получен и какой формат этого отклика.

Отклики могут быть текстовыми или двоичными. Если отклик текстовый, то обычно этот отклик в формате RFC откликов.

10.1.5 Переговоры (*conversations*)

Большинство протоколов аналогичны переговорам. Некоторые очень просты, а некоторые нет, но обычно они все равно в простом текстовом формате.

Переговоры означают следующую структуру:

1. отправка команды
2. возврат состояния
3. дополнительные данные отклика

Вернем обратно к примеру получения почтовых кодов, в котором обмен выглядит следующим образом:

```
Client: lookup 37642 77056
Server: 200 Ok
Server: 37642: CHURCH HILL, TN
Server: 77056: HOUSTON, TX
Server: .
```

Разделив на отдельные куски, разговор будет выглядеть так:

Команда:

```
Client: lookup 37642 77056
```

Ответ:

```
Server: 200 Ok
```

Отклик:

```
Server: 37642: CHURCH HILL, TN
Server: 77056: HOUSTON, TX
Server: .
```

Каждая из этих частей будет объяснена ниже.

10.2 RFC - определения

В настоящем нет действующих стандартов для терминов, обсуждаемых в данной главе. Тем не менее, эти термины базируются на квази-стандартах, на которых работают все тексто-ориентированые RFC протоколы.

Единственное значимое исключение- это протокол POP3. Это мистерия, почему разработчики решили идти своим путем, вместо пути RFC. Протокол POP3 в реальности очень ограничен и не предлагает нужной дополнительной функциональности к протоколу. Это обычная диверсия.

Протокол IMAP4, который был предложен как наследник POP3, продолжил эту порочную практику и так же использует не стандартный механизм. Протокол IMAP4 не получил широкого распространения и POP3 остается стандартным протоколом для почты.

После этих двух упоминаний, текстовый протокол все еще остается квази-стандартным. Этот квази-стандарт общий путь посылки команды и получение ответов и откликов.

10.2.1. RFC - коды состояния

Коды состояния RFC имеют следующую форму:

```
XXX Status Text
```

Где XXX это цифровой код в диапазоне 100-599.

Трехзначный цифровой номер означает ответ, который в ран-тайм, служит для определения результата выполнения команды. Обычно присутствует дополнительный текст, показываемый пользователю или для отладки. В этом случае, обычно используется английский язык, но может быть и локализован, если будет удовлетворять 7-битному коду ASCII. В некоторых случаях,

когда данные короткие, данные возвращаются в дополнительном текстовом поле. В этих случаях данные все рано должны быть 7-бит ASCII, но протокол сам определяет язык и ограничения форматов. В большинстве случаев, эти данные языко-независимы (*language neutral*). Например, команда "TIME", возвращает "15:30" в дополнительном текстовом поле.

Пример RFC ответа:

```
404 No file exists
```

404 это цифровой отклик и "No file exists" это дополнительное текстовое сообщение. Только код 404 должен быть интерпретирован программой и код 404 должен быть точно определен протоколом именно для этого. Текстовое сообщение обычно используется для отладки, ведения логов или для показа пользователю. Текстовое сообщение может иметь различные формы, от реализации к реализации и может быть локализировано на другие языки. Языки которые не соответствуют требованиям кода 7-бит ASCII должны быть транслитеризированы в английские символы.

Цифровая часть может быть назначена любым значениям. Тем не менее существует общее соглашение:

- 1xx - информационные
- 2xx - успешные
- 3xx – временные ошибки
- 4xx – постоянные ошибки
- 5xx – внутренние ошибки

Числа обычно уникальные, но не всегда. Если вы назначите 201 для "File not found", многие команды могут отвечать этим кодом и значение всегда одно. В некоторых редких случаях, значение числа зависит от примененной команды. В этом случае, каждая команда назначает специфическое значение each command assigns specific meanings to each numeric reply.

Цифровые коды, заканчивающие на 00, то есть 100, 200, и так далее резервированы для общих ответов, которые не имеют специального значения связанного с ними. 200 наиболее часто с "Ok".

Коды состояния могут находиться на нескольких строках и содержать большие текстовые сообщения. В этом случае, остальные строки, за исключением последней должны содержать код символа тире сразу после номера, вместо пробела.

Пример такого ответа:

```
400-Unknown Error in critical system
400-The server encountered an error and has no clue what caused it.
400-Please contact Microsoft technical support, or your local
400-tarot card reader who may be more helpful.
400 Thank you for using our products!
```

10.2.1.1. Примеры

Здесь пример некоторых кодов состояния, полученные от HTTP протокола. Как вы видите они относятся к разным категориям по первой цифре.

```
200 Ok
302 Redirect
404 Page not found
```

```
500 Internal Error
```

Если вы видите *500 Internal Error*, велик шанс, что вы используете Microsoft IIS. (Грубый наезд :-))

10.2.2. RFC – ответ (r*eply*)

RFC ответ, возвращает код состояния.

10.2.3. RFC – отклик (r*esponse*)

RFC отклик это текстовый ответ, ограниченный строкой с одной точкой в начале. Если данные содержат строку, которая содержит точку, то она перекодируется в две точки перед передачей, и обратно преобразовываться в одну при приеме.

RFC отклики очень применимы, когда заранее неизвестно количество возвращаемых данных. Это используется в HTTP, Mail, News и других протоколах.

Методы Indy, которые поддерживают RFC отклики – это Capture (для приема) и WriteStrings (для передачи).

10.2.4. RFC - транзакции

Транзакция RFC это общение, которое состоит из команды, ответа и дополнительного отклика, все в формате RFC. Обработчики команд и другие части Indy построены на транзакциях RFC.

Примет транзакции:

```
GET File.txt
201 File follows
Hello,
Thank you for your request, however we cannot grant your
request for funds for researching as you put it in your
application "A better mouse trap".
Thank you, and please do not give up.
.
```

10.3. Класс TIdRFCReply

TIdRFCReply обладает возможностью посылки и приема RFC ответов. TIdRFCReply имеет три основных свойства: NumericCode, Text и TextCode. NumericCode и TextCode взаимно исключающие. TextCode – свойство, которое управляет такими протоколами, как POP3 и IMAP4.

Для генерации ответа, установите свойство NumericCode (или TextCode) и дополнительно введите текст в свойство Text. Свойство Text типа TStrings позволяет многострочные ответы.

TIdRFCReply имеет методы для записи форматированных ответов, и также для разбора текста в TIdRFCReply.

TIdRFCReply используется для посылки ответов на команды, и также для свойств ReplyException, ReplyUnknown и Greeting properties класса TIdTCPServer.

10.4. Класс ReplyTexts

Цифровой код в ответе обычно уникален для каждой ошибки. Например, протокол HTTP использует код 404 для "Resource not found". Многим командам разрешено возвращать код 404 как ошибку, но код 404 всегда должен означать одну и туже ошибку. Для преодоления дублирования текстов для ошибки 404 класс TIdTCPServer имеет свойство ReplyTexts.

Свойство ReplyTexts – это коллекция экземпляров TIdRFCReply, которые могут быть обработаны, как в ран-тайм, так и в дизайн-тайм. Свойство ReplyTexts используется для обработки списка текстов, которые связаны с цифровым кодом. Когда свойство TIdRFCReply используется в TCPServer, который имеет цифровой код, но не имеет текстовой части, Indy просматривает в ReplyTexts и использует строку от туда.

Вместо включения текста, в каждый ответ 404 посмотрите ниже:

```
ASender.Reply.SetReply(404, 'Resource Not Found');
```

Затем может использоваться следующий код:

```
ASender.Reply.NumericCode := 404;
```

Перед тем, как Indy посылает ответ, она сначала устанавливает свойство Text из найденного в ReplyTexts. Это позволяет хранить все тексты ответов в одном месте, позволяя легко управлять ими.

10.5. Курица или яйцо?

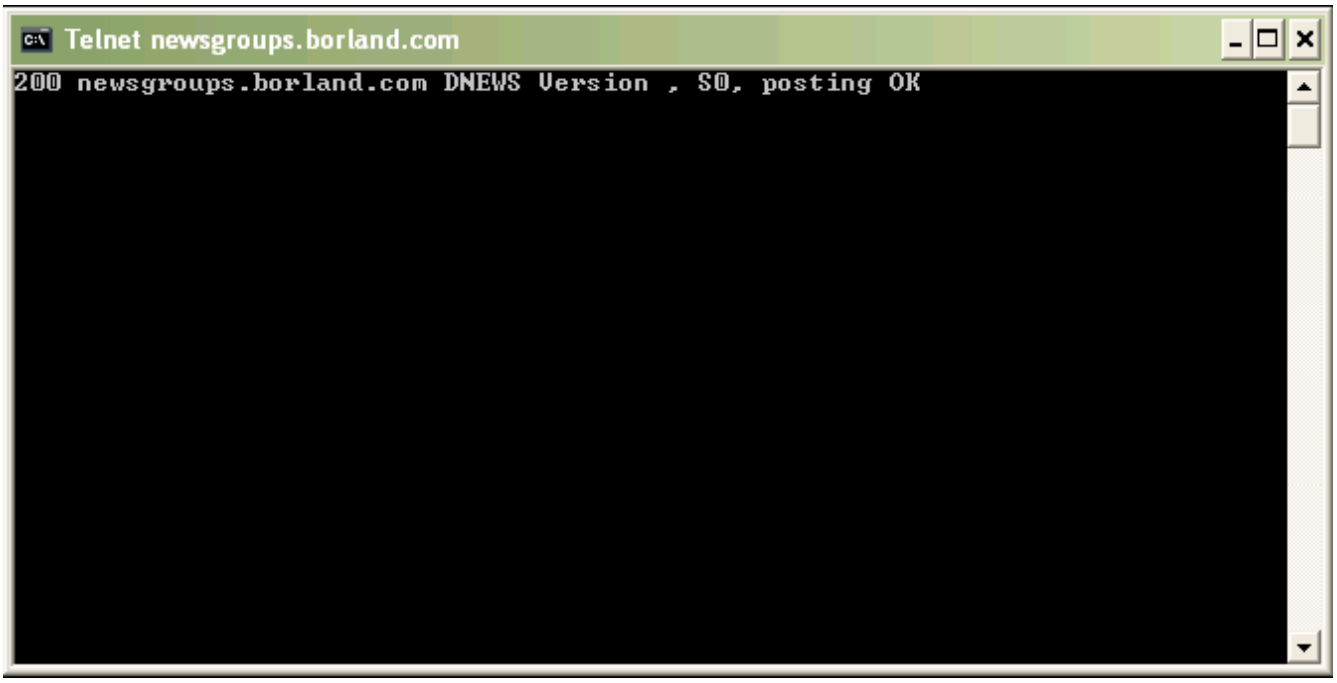
При построении системы, в которой вы должны построить, и сервер, и клиента, возникает следующий вопрос - "что делать сначала, клиента или сервера?". Оба нужны одновременно для отладки.

Ответ прост, проще построить сервер первым. Для тестирования клиента, вы должны иметь сервер. Для тестирования сервера вам нужен клиент. Но поскольку, протоколы как правило текстовые, то клиент легко эмулировать с помощью телнета.

Для тестирования, подсоединитесь к серверу на его порт. В командной строке введите:

```
Telnet newsgroups.borland.com 119
```

Теперь нажмите клавишу enter. После соединения вы должны увидеть экран, подобный нижнему рисунку.



На Windows 95/98/ME и NT это может выглядеть немного иначе, чем на Windows 2000 или на Windows XP, но результат должен быть таким же. Другие версии Windows загружают телнет,

как новое приложение в свом окне. Выше показанный рисунок относится к Windows XP', в котором телнет является консольным приложением.

Команда "Telnet newsgroups.borland.com 119" указывает клиенту телнета, подсоединиться к newsgroups.borland.com на порт 119. Порт 119 используется для протокола NNTP (News). Подобно подсоединения к серверу новостей Borland, телнет может использоваться для соединения с любым сервером, использующим текстовый протокол.

Для отсоединения от сервера новостей введите команду "Quit" и нажмите enter.

10.6. Определение пользовательского протокола

Задача сетевого разработчика состоит не только во взаимодействии с существующими системами, но часто и в создании собственных. В таком случае требуется создание своего протокола.

Первым шагом построения клиента или сервера – это разработка протокола. Для стандартных протоколов, это решается изучением соответствующего RFC. Если же протокол не стандартный, или уже определен, то должен быть определен.

При определении протокола, должны быть сделаны следующие решения:

- Текстовые или двоичные команды? Пока нет особых требований, используйте текстовые команды. Текстовые команды проще для понимания и для отладки.
- TCP или UDP? Это определяется от требований к протоколу. Изучите все характеристики и решите с осторожностью. В большинстве случаев TCP правильный выбор.
- Порт – каждому серверному приложению требуется выделенный порт для прослушивания. Порты ниже 1024 резервированы и никогда не должны использоваться, кроме реализации протокола, которому уже назначен порт ниже 1024.

После определения команд, также должны быть определены ответы и отклики.

10.7. Симуляция другой стороны (*Peer Simulation*)

Традиционно естественный путь построения клиента и сервера – это сначала построение сервера, а затем клиента или построения их параллельно. Но Indy имеет возможность построить клиент или сервер, отдельно друг от друга. В некоторых случаях, один может быть построен без необходимости доступа к другому. В таких случаях может использоваться эмуляция (видимо симуляция, судя по названию главы) другой стороны. Эмуляция другой стороны будет обсуждена позже в главе 14. Отладка.

10.8. Протокол получения почтового кода

В данной главе более подробно обсуждается Протокол получения почтового кода, который был представлен ранее в его клиенте. Сам сервер будет обсужден позже.

Проект был разработан так, чтобы быть как можно более простым. Сервер получения почтового кода (Zip код в Америке) позволяет клиенту запросить город и штат, к которому относится почтовый код.

Примерные данные, которые использует сервер, относятся к американским почтовым кодам, которые у них называются zip коды. Протокол может обрабатывать и другие коды, но американские коды уже заложены в демо программу.

Для тех, кто находится за пределами Соединенных Штатов Америки и не знаком с их системой, zip это почтовый код в США, который указывает регион доставки. Zip коды цифровые и состоят из 5 цифр. Zip коды могут также содержать дополнительные четыре цифры в формате 16412-0312. Данный тип кодов получил название Zip+4. Четыре дополнительные цифры уточняют локальную зону доставки и не требуются для определения города.

При создании протокола были приняты следующие решения:

- Все команды текстовые
- Транспорт TCP
- Порты: 6000. Порт 6000 это наиболее часто используемый номер в тестовых примерах Indy. Это не имеет значения.

Протокол поддерживает следующие команды

- Help – получение справки
- Lookup <почтовый код 1> < почтовый код 2> ... – запрос информации
- Quit – отсоединение от сервера

При разработке протокола, полезно знать ключевые протоколы, такие как - NNTP, SMTP и HTTP и использовать их как модель. Исключая POP3 и IMAP4. Поскольку это плохой пример построения протоколов.

Поскольку NNTP поддерживает передачу и прием сообщений в рамках одно протокола, Протокол NNTP будет упоминаться в данной книге несколько раз.

10.8.1. Команда Help

Команда Help очень часто используемая команда и редко применяемая в автоматических клиентах. Данная команда наиболее пригодна для использования человеком, который или тестирует, или напрямую работает с сервером. Почти все серверы реализуют данную команду.

Команда Help полезна для вывода справки о командах сервера и их возможном применении.

Вот пример ответа Борландовского сервера новостей сервера, на команду HELP:

```
help
100 Legal commands
authinfo user Name|pass Password
article [MessageID|Number]
body [MessageID|Number]
check MessageID
date
group newsgroup
head [MessageID|Number]
help
ihave
last
list [active|active.times|newsgroups|subscriptions]
listgroup newsgroup
mode stream
mode reader
newgroups yymmdd hhmmss [GMT] [<distributions>]
newnews newsgroups yymmdd hhmmss [GMT] [<distributions>]
next
post
```

```
slave
stat [MessageID|Number]
takethis MessageID
xgtitle [group_pattern]
xhdr header [range|MessageID]
xover [range]
xpat header range|MessageID pat [morepat...]
.
```

Для нашего протокола, сервер отвечает кодом 100, плюс сопроводительный текст.

10.8.2. Команда Lookup

Команда lookup принимает один или несколько почтовых кодов для поиска и возвращает название города и штат. Данные возвращаются в формате RFC откликов. Если код не найден, то отклик не возвращается (но если судить по примеру это не так, возвращает пустой отклик – строка с точкой). Код ответа "200 Ok".

Пример:

```
lookup 37642 16412
200 Ok
37642: CHURCH HILL, TN
16412: EDINBORO, PA
.
```

Даже если код не найден, то возвращается ответ "200 Ok".

```
lookup 99999
200 Ok
.
```

Мы приняли такое решение. Если бы сервер мог воспринимать только один параметр, то можно бы было отвечать кодом 200, и если не найден, то кодом 4XX. Но протокол может возвращать часть для правильных данных, поэтому было решено всегда возвращать код 200.

При частично правильных данных и ответ:

```
lookup 37642 99999
200 Ok
37642: CHURCH HILL, TN
.
```

Если бы протокол возвращал код ошибки, то частичные данные были бы проигнорированы. Данное решение позволило серверу отвечать и на частично правильные запросы, без генерации ошибки.

10.8.3. Команда Quit

Команда quit является прямым приказом серверу прекратить сессию и отсоединиться.

Посмотрим снова на сервер новостей Борланда. Его ответ следующий:

```
quit
205 closing connection - goodbye!
The postal code protocol responds similarly:
quit
201-Paka!      (Russians are everywhere ☺)
201 2 requests processed.
```

Почтовый протокол выдает многострочные ответы. Это не определено самим протоколом. Любые RFC ответы могут быть как однострочными, так и многострочными. Indy обрабатывает оба типа ответов автоматически.

11. Прокси (proxy – заместитель, уполномоченный)

Один из частых вопросов "Как я могу использовать Indy с прокси?". Когда задают этот вопрос, то ответить непросто. Не всегда возможно ответить, поскольку существует множество, самых различных прокси. Некоторые протоколы, также имеют свои собственные методы общения с прокси. Рассмотрим наиболее распространенные виды прокси.

Прокси и файрволы выполняют похожие задачи, но имеют разные цели. Поскольку у них похожие задачи, они могут использоваться для выполнения функций друг друга и могут быть объединены в комбинацию прокси-файрвол.

Прокси могут быть разделены на две категории:

- Прозрачные
- Непрозрачные

11.1. Прозрачные прокси

Прозрачные прокси – это прокси, которые не требуют вмешательства в протокол обмена. О наличии прозрачного прокси часто не догадываются разработчики или пользователи (на то они и прозрачные и их нельзя обойти).

Хотя прозрачные прокси не оказывают влияния на клиента, они оказывают влияние на сервера. В большинстве случаев, серверы позади таких прокси скрыты от остального мира. Это позволяет иметь доступ к серверу с другой стороны прокси через его порты и эти порты туннелируются снаружи внутрь.

11.1.1. Туннелирование IP / Трансляция сетевого адреса (*NAT*)

IP маскирование ли трансляция сетевого адреса (NAT) в прокси позволяет всем исходящис соединеним быть прозрачными и не оказывать влияния на клиентов. Клиенты продолжают работать как обычно и не требуется их конфигурирование.

Microsoft Internet Connection Sharing работает по данному методу.

11.1.2. Маппирование портов / Туннели

Маппированый порт или туннелированный прокси работает путем создания туннелей через блокированный маршрут. Маршрут может быть заблокирован в сетевой конфигурации, или может быть мостом между двумя сетями, или может быть умышленно защищен файрволом внутренней сети.

Внутренняя сеть определятся как одна сторона сети в локальной сети и другой сети или внешней сети, к которой подключена внутренняя сеть. (Бр – фраза, не подлежащая к переводу, очень трудная к пониманию, принимайте как получилось. Если сказать своими словами – это часть сети отделенная от других сетей, как локальных, так и глобальных Почему бы так и не сказать?)

Поскольку маршрут блокирован, весь доступ возможен только через туннелирование порта. Порты назначаются на локальном сервере, который доступен извне. Этот сервер затем пересылает данные из и во внутреннею сеть. Недостатком маппированых портов является то, что маппированый порт маппируется на фиксированный удаленный узел и порт. Для таких протоколов, как почтовые и сервера новостей они определены заранее и работают нормально. Но для протоколов подобных HTTP данный способ не работает, поскольку удаленное место неизвестно до общения (речь про доступ изнутри).

Маппированные порты могут быть использованы также для маппирования портов снаружи во внутреннею сеть. Маппированые порты часто используются совместно с NAT прокси, для представления серверов во внешней сети.

11.1.3. FTP User@Site прокси

Для реализации FTP прокси имеются несколько методов. Основной тип FTP прокси называется User@site.

При использования метода User@Site, все FTP сессии подсоединяются к локальному прокси серверу. Прокси притворяется что он FTP сервер. Прокси сервер перехватывает и интерпретирует FTP запросы. Когда прокси запрашивает имя пользователя, то имя пользователя и нужный FTP посылаются в виде username@ftpsite. Прокси соединяется нужным FTP и перехватывает команды передачи.

Для каждой команды передачи, прокси динамически маппирует локальный порт для передачи данных и модифицирует информацию передачи, возвращаемую клиенту. FTP клиент контактирует с прокси вместо доступа к реальному FTP серверу. Из-за трансляции, FTP клиент не знает, что прокси является ненастоящим сервером.

Например, пусть дан FTP сайт - ftp.atozedsoftware.com и имя пользователя joe, а его пароль smith, то нормальная сессия выглядит так:

```
Host: ftp.atozedsoftware.com
User: joe
Password: smith
```

Если User@Site прокси существует и его имя corpproxy, то FTP сессия выглядит так:

```
Host: corpproxy
User: joe@ftp.atozedsoftware.com
Password: smith
```

11.2. Непрозрачные прокси

Непрозрачные прокси требуют вмешательства в используемые протоколы. Многие протоколы имеют встроенную поддержку для непрозрачных прокси.

11.2.1. SOCKS прокси

SOCKS – это прокси, которые не требуют изменений в протоколе высокого уровня, но работают на уровне TCP. Для протоколов, которые используют SOCKS прокси, программная часть должна быть реализована на уровне TCP.

Если программное обеспечение не поддерживает явно SOCKS прокси, то оно не может быть использовано для работы с SOCKS файрволом. Большинство популярного обеспечения, такое как – браузеры и ICQ поддерживают SOCKS, но прочее программное обеспечение как правило нет. Поэтому, SOCKS часто должен развертываться с внутренними серверами, маппированными портами, другими прокси или их комбинациями.

Для программного обеспечения, которое поддерживает SOCKS, вместо соединения с сервером назначения, сначала происходит соединение с SOCKS прокси. При этом передается запись с данными, содержащими сервер назначения и дополнительно данные аутентификации. SOCKS сервер затем динамически строит туннель к серверу.

Поскольку SOCKS протокол работает динамически на основе переданных данных, он очень настраиваемый и гибкий.

11.2.2. HTTP (CERN) прокси

HTTP прокси, иногда называемый CERN прокси, это специализированный прокси, который обслуживает только трафик браузеров. В дополнение HTTP, также может обслуживать и FTP трафик, если FTP делается поверх HTTP. HTTP прокси могут также кэшировать данные и часто используется только по этой причине.

Многие корпоративные сети часто закрывают свои внутренние сети файрволом и разрешают выход наружу только через HTTP и почту. Почта предоставляется с помощью внутреннего почтового сервера, а HTTP предоставляется с помощью HTTP прокси. Данный тип HTTP является дружественным файрволу, поэтому многие новые протоколы используют HTTP как транспорт. SOAP и другие web службы являются замечательным примером.

12. Обработчики ввода/вывода (IOHandlers)

Indy может настраиваться и расширяться многими путями, без необходимости напрямую модифицировать исходный код. Примером такой расширяемости являются обработчики ввода/вывода (*IOHandlers*). Обработчики ввода/вывода позволяют вам использовать любой источник ввода/вывода I/O в Indy. Обработчики ввода/вывода должны использоваться, когда вы желаете использовать альтернативный механизм ввода/вывода или создание нового транспортного механизма.

Обработчики ввода/вывода выполняют весь ввод/вывод (Input/Output) для Indy. Indy не выполняет ни какого своего ввода/вывода, за пределами обработчика ввода/вывода. Обработчик ввода/вывода используется для посылки сырых TCP данных для компонент Indy.

Обработчики ввода/вывода позволяют классам обрабатывать весь ввод/вывод в Indy. Обычно, весь ввод/вывод делается через сокет и обслуживается обработчиком по умолчанию - TIdIOHandlerSocket.

Каждый TCP клиент имеет свойство IOHandler, которое может быть назначено обработчику IOHandler, как это делает каждое серверное соединение. Если обработчик IOHandler не указан, то неявно используется TIdIOHandlerSocket, который создается автоматически и используется TCP клиентом. TIdIOHandlerSocket реализует ввод/вывод используя TCP сокет. Indy также включает дополнительные обработчики ввода/вывода: TIdIOHandlerStream и TIdSSLIOHandlerSocket.

Другие обработчики ввода/вывода могут быть созданы, позволяя Indy использовать любые источники ввода/вывода, которые вы только можете вообразить. В данный момент Indy поддерживает только сокеты, потоки и SSL как источники ввода/вывода, но источники ввода/вывода позволяют и другие возможности. Пока нет других планов, но обработчики ввода/вывода могут быть созданы для поддержки туннелей, IPX/SPX, RS-232, USB или Firewire. Indy не ограничивает вас в выборе вашего источника ввода/вывода и использование обработчиков ввода/вывода позволяет это сделать.

12.1. Компоненты IOHandler

12.1.1. Компонент TIdIOHandlerSocket

Компонент TIdIOHandlerSocket это обработчик IOHandler по умолчанию. Если обработчик не указан явно, то он создается неявно для вас. Компонент TIdIOHandlerSocket обрабатывает весь ввод/вывод, относящийся к TCP сокетам.

Обычно, компонент TIdIOHandlerSocket не используется явно, пока не потребуются расширенные способности.

12.1.2. Компонент TIdIOHandlerStream

Компонент TIdIOHandlerStream используется для отладки и тестирования. При его использовании все взаимодействие с сервером в TCP сессии может быть записаны. Позже вся сессия может быть «проиграна». Компоненты Indy не знают, работают ли они с реальным сервером и реальным соединением.

Это очень мощное средство отладки в дополнение к инструменту QA отладки. Если у пользователя есть проблемы, то специальная версия программы может быть послана пользователю или включены средства отладки для ведения лога сессии. Используя лог файлы, вы можете затем реконструировать сессию пользователя в локальной отладочной среде.

12.1.3. Компонент TIdSSLIOHandlerSocket

Компонент TIdSSLIOHandlerSocket используется для поддержки SSL. Обычно компрессия и декомпрессия данных может быть реализована с использованием перехватчиков (Intercepts) вместо IOHandlers. Но SSL библиотека используемая Indy (OpenSSL), работает напрямую с сокетом, вместо трансляции данных посылаемых Indy. Поэтому реализация выполнена как IOHandler. . TIdSSLIOHandlerSocket является наследником TIdIOHandlerSocket.

12.2. Пример - Speed Debugger

Пример Speed Debugger демонстрирует как симулировать медленное соединение. Это полезно, как для отладки, так и для тестирования ваших приложений при симуляции медленных сетей, таких как модемы.

Пример Speed Debugger состоит из главной формы и пользовательского обработчика IOHandler. Speed Debugger использует компонент «маппированый порт» для передачи данных конкретному web серверу. Браузер соединяется со Speed Debugger и Speed Debugger пробует получить страницу с указанного web сервера, затем возвращает ее web браузеру замедленно, с указанной скоростью.

Поле текстового ввода используется для указания web сервера.

Примечание: не указывайте URL и не указывайте протокол http:// или web страницу. Оно предназначено только для указания имени сервера или его IP адреса. Если у вас есть локальный web сервер, то вы можете просто указать 127.0.0.1 и использовать локальный web сервер.

Комбо-бокс используется для выбора скорости и состоит из следующих выборов. Симулируемая скорость указывается в скобках.

- Apache (Unlimited)
- Dial Up (28.8k baud)
- IBM PC XT (9600 baud)
- Commodore 64 (2400 baud)
- Microsoft IIS on a PIII-750 & 1GB RAM (300 baud) (Боже как же он его ненавидит! :-))

После нажатия кнопки Test - Speed Debugger загружает браузер по умолчанию с URL http://127.0.0.1:8081/. В данном случае браузер делает запрос из Speed Debugger. Speed Debugger слушает на порту 8081, который может конфликтовать с некоторым существующими локальными web серверами.

Пример Speed Debugger может быть загружен с Indy Demo Playground.

12.2.1. Пользовательский обработчик IOHandler

Работа Speed Debugger делается на основе пользовательского IOHandler. Маппированый порт компонент имеет событие OnConnect, и данное событие используется как хук в нашем IOHandler для каждого исходящего клиента, который создает маппированый порт. Это выглядит так:

```
procedure TformMain.IdMappedPortTCP1Connect(AThread: TIdMappedPortThread);
var
  LClient: TIdTCPConnection;
  LDebugger: TMyDebugger;
begin
  LClient := AThread.OutboundClient;
```

```
    LDebugger := TMyDebugger.Create(LClient);
    LDebugger.BytesPerSecond := GSpeed;
    LClient.IOHandler := LDebugger;
  end;
```

Пользовательский класс IOHandler, называется TMyDebugger и реализован как наследник от TIdIOHandlerSocket, являющегося наследником IOHandler. Поскольку TIdIOHandlerSocket уже реализует весь актуальный ввод/вывод, TMyDebugger должен только замедлить передачу данных. Это делается путем перекрытия метода Recv.

Из метода Recv вызывается наследованный Recv для приема данных. Затем, базируясь на выбранном ограничении скорости, рассчитывается необходимая задержка. Если рассчитанная величина больше, чем наследованная, то метод Recv вызывает Sleep. Это может казаться сложным, но на самом деле это очень просто. Метод Recv приведен ниже.

```
function TMyDebugger.Recv(var ABuf; ALen: integer): integer;
var
  LWaitTime: Cardinal;
  LRecvTime: Cardinal;
begin
  if FBytesPerSecond > 0 then
  begin
    LRecvTime := IdGlobal.GetTickCount;
    Result := inherited Recv(ABuf, ALen);
    LRecvTime := GetTickDiff(LRecvTime, IdGlobal.GetTickCount);
    LWaitTime := (Result * 1000) div FBytesPerSecond;
    if LWaitTime > LRecvTime then
    begin
      IdGlobal.Sleep(LWaitTime - LRecvTime);
    end;
  end
  else
  begin
    Result := inherited Recv(ABuf, ALen);
  end;
end;
```

13. Перехватчики (Intercepts)

Перехватчик – это более высокий уровень, чем обработчик ввода/вывода и используется для модификации или перехвата данных независимо от источника и приемника.

Перехватчики были сильно изменены в версии 9.0. В версии 8.0 перехватчик мог предоставлять только весьма ограниченную функциональность обработчика ввода/вывода. Перехватчики могут трансформировать уже принятые данные или данные для передачи. Перехватчики более не поддерживают никакой функциональности обработчика ввода/вывода (IOHandler), поскольку введены новые классы IOHandler, со всей необходимой функциональностью.

Перехватчики попрежнему могут выполнять преобразования данных и но предоставляют больше возможностей в Indy 9.0. Возможности трансформация в Indy 8.0 были сильно ограничены, так как данные должны были иметь один и тот же размер. Это делало невозможным использовать их для реализации сжатия данных или логирования, поскольку не позволяло изменить размер данных.

Перехватчики позволяют изменять данные после того, как они были приняты в IOHandler или изменять их перед посылкой в IOHandler. Перехватчики в данный момент используются для реализации ведения логов и отладки компонент. Они могут быть также использованы для реализации шифрования, сжатия, статических коллекторов или ограничитения трафика.

13.1. Перехватчики

Перехватчики взаимодействуют с входящими или исходящими данными и позволяют их записывать в лог или модифицировать. Перехватчики позволяют изменять входящие данные, после того как они приняты из сети, перед выдачей их пользователю.

Перехватчики также позволяют модифицировать исходящие данные перед посылкой в сеть. Перехватчики могут быть использоваться для реализации ведения логов, шифрования и сжатия данных.

Клиентские перехватчики базируются на соединениях (по одному на каждое). Они также могут быть использованы на сервере, если они назначены индивидуальному соединению.

Примечание: Перехватчики в Indy 9 отличаются от перехватчиков в Indy 8. Перехватчики в Indy 8, выполняли комбинированную роль перехватчика и обработчика ввода/вывода. Что делало сложным разделение функций перехвата и обработки. Перехватчики Indy 8 также не могли изменять размер данных и поэтому были непригодны для сжатия.

13.2. Ведение логов (*Logging*)

Indy 8.0 имел один компонент ведения логов, который мог быть использован для различных источников. В Indy 9.0 компоненты логов теперь базируются на новом общем классе и имеют специализированные классы. Базовый класс также предоставляет свойства и функциональность, такую как регистрация времени, в дополнение к данным.

Все классы ведения логов реализованы, как перехватчики. Это означает, что они перехватывают входящие данные, после того, как они были прочитаны и перед передачей исходящих в источник.

Специализированные классы логов, следующее:

- **TIdLogEvent** – возбуждает события, когда данные приняты, или переданы, или при появлении события состояния. Класс TIdLogEvent полезен для реализации пользовательских логов, без необходимости в реализации нового класса.
- **TIdLogFile** – Записывает данные в файл.
- **TIdLogDebug** – Записывает данные в окно отладки Windows или в консоль Linux. Также отмечает данные, как принятые данные, переданные данные или информация о статусе. Класс TidLogDebug полезен для проведения простой отладки.
- **TIdLogStream** – Не добавляет комментариев, отметок к данным, как другие классы. Вместо этого просто записывает сырые данные в указанный поток. Класс TIdLogStream может использоваться по разному, но обычно он очень эффективно используется для QA тестирования и удаленной отладки. Могут быть построены и пользовательские классы логов.

14. Отладка

Обычно отладка клиентов проще отладки серверов. Клиенты просто должны обслуживать только одно соединение и могут обычно быть отлажены с помощью простой техники отладки. В данной главе приводится несколько полезных советов для отладки клиентов и серверов.

14.1. Ведение логов

Реально простой путь – что увидеть, что делает клиент, без трассировки по коду – это использовать классы TIdLogDebug или TIdLogFile. Класс TIdLogDebug выводит информацию в напрямую в окно отладки и очень удобен для наблюдения, того, что клиент посылает и принимает в реальном масштабе времени. Если вы не желает смотреть трафик в реальном времени, то воспользуйтесь классом TIdLogFile. После того как клиент закончит свою работу, вы можете посмотреть содержимое файла и увидеть, что делалось во время сессии.

14.2. Симуляция

Но иногда требуется симулировать клиента или сервер, если он не доступен. Это может быть сделано по причинам безопасности, трафика или по другим причинам. В таких случаях вы можете воспользоваться симуляцией. Симуляция может также использоваться для построения клиента, до построения сервера или для создания тестовых скриптов, путем симуляции клиента.

Симуляция может быть выполнена с помощью TIdIOHandlerStream и назначения потока вывода в текстовый файл, поток ввода должен быть установлен в nil. Это указывает классу TIdIOHandlerStream читать все данные для посылки клиенту из текстового файла и игнорировать все данные возвращаемые с клиента.

Если вы назначите поток ввода, то будет происходить ведение лога с данными клиента.

14.3. Запись и воспроизведение

Одним из полезных методов является запись сессии, и последующего ее воспроизведения. Это особо востребовано для регрессивного тестирования и удаленной отладки. Если ваш заказчик находится в удаленном месте, то вы можете послать ему особую версию программы или включить запись сессии. Затем они могут вам переслать записанный файл, и вы сможете симулировать его сессию клиента или сервер у себя на месте, без необходимости иметь реальное соединение с их сервером.

Для выполнения этого используйте класс TIdLogStream для записи принятых данных в файл. Вы сможете также записать данные, которые клиент передавал в отдельный файл, но вы не нуждаетесь в этом, пока не потребуется их посмотреть вручную. После того, как вы получите файл, вы можете его подсоединить к компоненту TIdIOHandlerStream.

Обработчики ввода/вывода (IOHandlers) также имеют другое полезное применение. Они используются для записи и воспроизведения. Живая сессия может быть записана с помощью компонентов ведения логов и позже воспроизведена с помощью потоков обработчиков ввода/вывода. Представим, что вы имеет заказчика, у которого есть проблемы, но вы не можете воспроизвести эти проблемы у себя и не можете посетить его. Вы можете попросить его прислать лог полной сессии и попытаться воспроизвести сессию на вашей машине. Команда Indy использует это как часть своей QA отладки. Я планирую описать это позже.

15. Параллельное выполнение (*Concurrency*)

В многопоточной среде ресурсы должны быть защищены так, чтобы они не были разрушены при одновременном доступе.

Параллельное выполнение и потоки переплетаются и выбор, что изучать сначала может быть сложным. В данном разделе мы рассмотрим параллельное выполнение и попытаемся дать понятия для изучения потоков.

15.1. Терминология

15.1.1. Параллельное выполнение

Параллельное выполнение – это состояние, когда много задач возникает в одно и тоже время. Когда конкурирование реализовано правильно это может рассматривать как «гармония», а если плохо, то как «хаос» .

В большинстве случаев, задача выполняется в потоке (thread). Но также это могут быть процессы (process) или волокна (fiber). Разделение очень условное, но использование правильной адстракции является ключем к успеху.

15.1.2. Борьба (споры) за ресурсы (*Contention*)

Что же такое точно борьба за ресурсы? Борьба за ресурсы – это когда более чем одна задача пытается получить доступ к одному и тому же ресурсу, в то же самое время.

Для тех, кто живет в большой семье обычно понятно, о чем идет речь, и может дать пример споров. Представим себе семью с шестью маленькими детьми, когда мама ставит на сто маленькую пиццу. Будет драка.

Когда множеству конкурирующих задач требуется доступ к данным режиме чтения/записи, то данные должны быть защищены. Если доступ не контролируется, две или более задач могут разрушить их, когда одна задача пытается писать, а другая читать одновременно. Если одна задача записывает во время чтения другой, то возможно чтение и запись несогласованных данных.

Обычно в этом случае не возникает исключения, и ошибка проявится позже в программе.

Проблемы борьбы за ресурсы не возникают в приложениях при малой нагрузке и поэтому не обнаруживают себя во время разработки. По этому правильное тестирование должно производиться на большой нагрузке. Иначе это подобно «русской рулетке” и проблемы будут возникать редко во время разработки и часто в рабочей среде.

15.1.3. Защита ресурсов (*Resource Protection*)

Защита ресурсов – это способ разрешения проблемы борьбы за ресурсы. К функциям защиты ресурсов относится разрешение доступа только одной задаче в одно и то же время.

15.2. Разрешение споров (*Resolving Contention*)

Когда множеству потоков требуется доступ до ресурсов в режиме чтения/записи, данные должны контролироваться для защиты их целостности. Это может наводить ужас на программиста не умеющего работать с потоками.

Как правило, большинство серверов не нуждаются в глобальных данных. Обычно им требуется читать данные только во время инициализации программы. Так как здесь нет доступа по записи потоки могут читать глобальные данные без побочных эффектов.

Общи пути разрешения споров приведены ниже.

15.2.1. Только чтение (*Read Only*)

Самый простой метод – это режим только чтение. Все простые типы (integers, strings, memory), которые используются в режиме только чтения - не требуют защиты. Это также относится и к таким сложным типам как TList и другие. Классы безопасны если только не используют глобальные переменные или глобальные поля классов в режиме чтения/записи.

В дополнение, ресурсы должны быть записаны перед любыми попытками чтения. Это делается путем инициализацией ресурсов во время запуска, до того как задачи будут к ним обращаться.

15.2.2. Атомарные операции (*Atomic Operations*)

Методология утверждает, что атомарные операции не требуют защиты. Атомарные операции – это такие операции, которые очень малы для деления в процессоре. Поскольку, их размер маленький, то это не приводит к спорам, так как они выполняются разом и не могут быть прерваны во время исполнения. Обычно атомарные операции – это строки кода компилируемые в одну иструкцию ассемблера.

Обычно задачи, такие как чтение или запись целых чисел (integer) или логических полей (boolean field) являются атомарными операциями, так как они выполняются с помощью единственной инструкции move. Тем не менее моя рекомендация, такая – никогда не рассчитывайте на атомарность операций, поскольку в некоторых случаях, даже при записи целого (integer) или логического (Boolean) может быть выполнено более одной инструкции, в зависимости от того, откуда данные читались. В дополнение, это требует знания внутренней природы компилятора, которое является предметом для изменения в любое время, без вашего оповещения. Закладываясь на атомарные операции вы можете написать код, который будет неоднозначным в некоторых случаях и может работать по разному на многопроцессорных машинах или других операционных системах.

Я до недавнего времени считал атомарные операции очень защищенными. Тем не менее, приход .NET принесет серьезную проблему. Когда наш код был компилирован в IL, и затем перекомпилирован в машинный код на платформах различных производителей, можете ли вы быть уверены, что каждая строка вашего кода будет атомарной операцией в конце концов?

Выбор конечно ваш, и конечно есть аргументы как за, так и против атомарных операций. Атомарные операции в большинстве случаев сохраняют только несколько микросекунд и несколько байт кода. Я настоятельно рекомендую не использовать атомарные операции, так как они дают мало преимуществ и являются потенциальным источником неприятонстей .

15.2.3. Поддержка Операционной Системы (*Operating System Support*)

Многие операционные системы имеют поддержку базовых потоко-безопасных операций.

Windows поддерживает набор функций, известных как, блочные (Interlocked) функции. Польза от этих функций очень маленькая и состоит в поддержке только простой функциональности для целых чисел, как увеличение, уменьшение, добавление, обмен и обмен со сравнением.

Количество функций варьируется от версии Windows и может быть причиной замораживания (deadlocks) в старых версиях Windows. В большинстве приложений они имеют малые плюсы.

Поскольку их функциональность ограничена, не поддерживается повсюду, и дает лишь некоторое увеличение производительности, вместо них в Indy рекомендуется использовать потоко-безопасные эквиваленты (заменители). (threadsafe).

Windows также имеет поддержку объектов IPC (interprocess communication), обертка вокруг которых имеется в Delphi. Данные объекты особенно полезны для программирования потоков и IPC.

15.2.4. Явная защита (*Explicit Protection*)

Явная защита заставляет каждую задачу знать, что ресурсы защищены и требуют явных шагов по доступу к ресурсу. Обычно такой код расположен в отдельной подпрограмме, которая используется многими задачами конкурентно и работает как потоко-безопасная обертка.

Явная защита обычно использует для синхронизации доступа к ресурсу специальный объект. Вкратце, защита ресурса ограничивает доступ к ресурсу для одной задачи за раз. Защита ресурса не ограничивает доступ к ресурсу, это требует знания специфики каждого ресурса. Вместо этого она подобна сигналам движения и код адаптирован так чтобы следовать и управлять сигналам трафика. Каждый объект защиты ресурса реализует определенный тип сигнала, используя различную логику управления и налагая различные ограничения на производительность . Это позволяет выбрать подходяший объект защиты в зависимости от ситуации.

Существует несколько типов объектов защиты ресурсов и они будут отдельно рассмотрены позже.

15.2.4.1. Критические секции (*Critical Sections*)

Критические секции могут быть использованы для управления доступом к глобальным ресурсам. Критические секции требуют мало ресурсов и реализованы в VCL как TCriticalSection. Вкратце, критические секции позволяют отдельному потоку в многопоточном приложении, временно блокировать доступ к ресурсу для других потоков исаользующих эту же критическую секцию. Критические секции подобны светофорам, которые зажинают зеленый сигнал только если впереди нет ни одной другой машины. Критические секции могут быть использованы, чтобы быть уверенным, что только один поток может исполняться в одно время. Поэтому, защищенные блоки должны быть минимального размера, так как они могут сильно понизить производительность, если будут неправильно использованы. Поэтому, каждый уникальный блок должен также использовать свою собственную критическую секцию (TCriticalSection) вместо использования единой для всего приложения.

Для входа в критическую секцию вызовите метод Enter и затем метод Leave для выхода из секции. Класс TCriticalSection также имеет методы Acquire и Release, которые аналогичны Enter и Leave соответственно.

Предположим, что сервер должен вести лог клиентов и отображать информацию в главном потоке. Мнение, что это должно быть синхронизироваться. Тем не менее, использование данного метода может негативно сказаться на производительности потока соединения, если много клиентов будут подключаться одновременно. В зависимости от потребностей сервера, лучшим решением был бы лог информации, а главный поток мог бы читать ее по таймеру. Следующий код является примером данной техники, которая использует критические секции.

```
var
  GLogCS: TCriticalSection;
  GUserLog: TStringList;

procedure TformMain.IdTCPServer1Connect(AThread: TIdPeerThread);
```

```
var
  s: string;
begin
  // Username
  s := ReadLn;
  GLogCS.Enter;
  try
    GUserLog.Add('User logged in: ' + s);
  finally
    GLogCS.Leave;
  end;
end;

procedure TformMain.Timer1Timer(Sender: TObject);
begin
  GLogCS.Enter;
  try
    listbox1.Items.AddStrings(GUserLog);
    GUserLog.Clear;
  finally
    GLogCS.Leave;
  end;
end;

initialization
  GLogCS := TCriticalSection.Create;
  GUserLog := TStringList.Create;

finalization
  FreeAndNil(GUserLog);
  FreeAndNil(GLogCS);
end.
```

В событии Connect имя пользователя читается во временную переменную перед входом в критическую секцию. Это сделано, чтобы избежать блокирования кода низкоскоростным клиентом в критической секции.

Это позволяет сетевому соединению быть выполненным до входа в критическую секцию. Для сохранения максимальной производительности, код в критической секции сделан минимально возможным.

Событие Timer1Timer возбуждается в главной форме. Интервал таймера может быть короче для более частых обновлений, но потенциально может замедлить восприятия соединения. Если требуется выполнять логирование и в других местах, кроме регистрации пользователей, то существует большая вероятность появления узкого места в производительности. Больший интервал обновлений, сводит задержки в интерфейсе к минимуму. Конечно, многие серверы не имеют никакого интерфейса, а те в которых он есть, он является вторичным и выполняется с меньгим приоритетом, чем поток, обслуживающий клиентов, что вполне допустимо.

Примечание пользователям Delphi 4: Класс TCriticalSection находится в модуле SyncObjs. Модуль SyncObjs обычно не включен в Delphi 4 Standard Edition. Если Вы используете Delphi 4, то Вы можете загрущить SyncObjs.pas файл с web сайта Indy. Этот файл не срдержит всей функциональности реализованной Борланд, но имеет реализацию класса TCriticalSection.

15.2.4.2. Класс TMultiReadExclusiveWriteSynchronizer (*TMREWS*)

В предыдущем примере, Класс TCriticalSection был использован для защиты доступа к глобальным данным. Он нужен случаях когда глобальные данные всегда обновляются. Конечно,

если глобальные данные должны быть доступны в режиме чтения и только иногда для записи, то использования класса TMultiReadExclusiveWriteSynchronizer будет более эффективно.

Класс TMultiReadExclusiveWriteSynchronizer имеет очень длинное и трудно читаемое имя. Поэтому мы будем называть его просто TMREWS.

Преимущества использования TMREWS состоит в том, что он позволяет конкурентное чтение из многих потоков, и действует как критическая секция, позволяя только одному потку доступ для записи. Недостатком TMREWS является, что он более сложен в использовании.

Вместо Enter/Acquire и Leave/Release, TMREWS имеет методы: BeginRead, EndRead, BeginWrite и EndWrite.

15.2.4.2.1. Специальное примечание к классу TMREWS

До Delphi 6 в классе TMultiReadExclusiveWriteSynchronizer имелась проблема, приводящая к взаимному блокированию (dead lock) при повышении уровня блокировки с чтения на запись. Поэтому, вы не никогда должны использовать данную возможность изменения блокировки чтения в блокировку записи, несмотря на то, что документация утверждает что это можно сделать.

Если вам нужна подобная функциональность, то имеется обходной путь. Он состоит в том, что сначала надо освободить блокировку чтения, а затем поставить блокировку записи. Тем не менее, если вы установили блокировку записи, то вы должны затем снова проверить условие, необходимое для начала записи. Если оно все еще выполняется, делаете свою запись, в противном случае немедленно снимите блокировку.

Класс TMultiReadExclusiveWriteSynchronizer так требует особой осторожности при использовании в Delphi 6. Все версии класса TMultiReadExclusiveWriteSynchronizer включая, поставляемый в update pack 1 и в update pack 2 имеют серьезные проблемы, которые могут вызвать взаимную блокировку. Обходных путей пока нет.

Borland в курсе этого и выпустил неофициальный патч и также ожидаются официальные патчи.

15.2.4.2.2. Примечание к классу TMREWS в Kylix

Класс TMultiReadExclusiveWriteSynchronizer в Kylix 1 и Kylix 2 реализован с помощью критических секций и не имеет преимуществ перед критическими секциями. Это сделано, что бы можно было писать общий код и для Linux и для Windows.

В будущих версиях Kylix, класс TMultiReadExclusiveWriteSynchronizer вероятно будет изменен, что бы работал также как Windows.

15.2.4.3. Выбор между Critical Sections и TMREWS

Поскольку класс TMREWS имеет некоторые проблемы, мой совет просто избегать его. Если вы решили использовать его, вы должны быть уверены, что это действительно лучший выбор и он должен быть реализован с помощью пропатченной версии, не подверженной зависаниям.

Правильное использование TCriticalSection в большинстве случаев дает обычно такой же быстрый синхронизированный доступ, а в некоторых случаях самый. Научитесь правильно использовать TCriticalSection, так как неправильное исаольщование может иметь негативное влияние на производительность.

Ключом к защите любых ресурсов является использование множества точек входа в секции и выполнение критических секций кода как можно быстрее. Когда проблема может быть

разрешена с помощью критических секций, то она должна решаться с их помощью, вместо использования TMREWS, поскольку критические секции проще и быстрее. В общем, всегда используйте критические секции вместо TMREWS.

Класс TMREWS работает лучше, если встретятся следующие условия:

1. доступ осуществляется по чтению и записи.
2. преобладает доступ по чтению.
3. период блокировки велик и не может быть разбит на меньшие независимые куски.
4. доступен пропатченый класс TMREWS и известно что он работает корректно.

15.2.4.4. Сравнение производительности

Как уже упоминалось ранее критические секции легковесны и мало влияют на производительность. Они реализованы в ядре операционной системы. ОС реализует их используя короткие эффективные ассемблерные команды.

Класс TMREWS более сложен и поэтому больше влияет на производительность. Он должен управлять списком запросов для поддержания состояния блокировок.

Для того чтобы продемонстрировать разницу был создан демонстрационный проект ConcurrencySpeed.dpr. Он проводит три простых замера:

1. TCriticalSection – Enter и Leave
2. TMREWS – BeginRead и EndRead
3. TMREWS – BeginWrite и EndWrite

Он делает это выполняя цикл заданное количество раз. Для примера 100000. В моих тестах я получил следующие результаты.

```
TCriticalSection: 20
TMREWS (Read Lock): 150
TMREWS (Write Lock): 401
```

Конечно, результаты зависят от компьютера. Но важна разница, а не абсолютные числа. Я могу видеть что при оптимальных условиях запись TMREWS в 7.5 раз медленне критических секций. А запись медленнее в 20 раз.

Нужно также заметить, что критические секции практически не деградирую при нагрузке, тогда как TMREWS сильно сдает. Тест выполнялся в простом цикле, и не было других запросов на блокировку. В реальной жизни TMREWS будет еще медленнее чем показано здесь.

15.2.4.5. Мьютексы (*Mutexes*)

Функции мьютексов почти полностью аналогичны критическим секциям. Разница состоит в том, что мьютексы - это более мощная версия критических секций с большим количеством свойств и конечно в связи с этим большим воздействием на производительность.

Мьютексы имеют дополнительные возможности по именованию, назначению атрибутов безопасности и они доступны между процессами.

Мьютексы могут быть использованы для синхронизации потоков, но они редко используются в данном качестве. Мьютексы были разработаны и используются, для синхронизации между процессами.

15.2.4.6. Семафоры (*Semaphores*)

Семафоры подобны мьютексам, но вместо единичного доступа, позволяют множественный. Количество доступов, которое разрешено определяется при создании семафора.

Представим, что мьютекс это охранник в банке, который позволяет доступ только одному человеку к банкомату (ATM). Только один человек за раз может использовать его и охранник защищает машину от доступа нескольких человек одновременно.

В данном случае, семафор будет более предпочтителен, если установлено четыре банкомата. В этом случае охранник позволяет нескольким людям войти в банк и использовать эти банкоматы, но не более четырех человек одновременно .

15.2.4.7. События (*Events*)

События – это сигналы, которые могут быть использованы между потоками или процессами, для оповещения о том, что что-то произошло. События могут быть использованы для оповещения других задач, когда что-то произошло или требуется вмешательство.

15.2.5. Потоко-безопасные классы

Потоко-безопасные классы были специально разработаны для защиты специфических типов ресурсов. Потоко-безопасные классы реализуют специфический тип ресурса и имеют сокровенные знания, что это за ресурс и как он функционирует.

Потоко-безопасные классы могут быть простыми, как потоко-безопасный integer или комплексными, как потоко-безопасные базы данных. Потоко-безопасные классы внутри используют потоко-безопасные объекты для выполнения своих функций.

15.2.6. Изоляция (*Compartmentalization*)

Изоляция – это процесс изоляции данных и назначения их только для использования одной задачей. На серверах изоляция - это естественный путь, так как каждый клиент может обслуживаться выделенным потоком.

Когда изоляция не является естественной, необходимо оценить возмодность ее использования . Изоляция часто может быть достигнута путем создания копии глобальных данных, работы с этими данными и затем возврата этих данных в глобальную область. При использовании изоляции, данные блокируются только во время инициализации и после окончания выполнения задачи, или во время применения пакетных обновлений.

16. Кодовые потоки

Многопоточность страшит многих программистов и часто расстраивает новичков. Потоки это элегантный путь решения многих проблем и он, однажды освоенный станет ценных вложением в вашу копилку опыта. Тема потоков может потребовать написания отдельной книги.

16.1. Что такое поток?

Поток это ваш код исполняемый паралелльно с другим вашим кодом в одной программе. Использование потоков позволяет выполнять несколько задач одновременно.

Представим, что в вашей компании только один телефон. Поскольку, имеется только одна телефонная линия и только один человек может использовать ее в одно время. Тем не менее, если в установите несколько телефонных линий, то и другие смогут делать телефонные звонки не узнавая свободна линия или нет. Потоки позволяют вашему приложению делать несколько дел одновременно.

Параллельное выполнение потоков доступно даже если у вас один процессор. В действительности только одни поток исполняется, но операционная система принудительно прерывает потоки и исполняет их по очереди. Каждый поток выполняется только в течение очень короткого интервала времени. Это позволяет делать десятки тысяч переключений в секунду. Поскольку, переключение, вытесняющее и непредсказуемое, то будет казаться, что программы выполняются параллельно и программное обеспечение должно быть предусмотрено для работы в таком режиме.

Если процессоров несколько, то потоки реально могут выполняться параллельно, но все равно каждый процессор выполняет только один поток.

16.2. Достоинства потоков

Использование потоков дает дополнительные преимущества перед обычным однопоточным дизайном.

16.2.1. Управление приоритетами (*Prioritization*)

Приоритеты отдельных потоков могут быть подстроены. Это позволяет отдельным серверным соединениям или клиентам получать больше процессорного времени.

Если вы повышаете приоритет всех потоков, эффект не будет заметен, так как все они будут иметь равный приоритет. Тем не менее они могут отнять время у потоков других процессов. Вы должны осторожно относиться к этому и не устанавливать слишком высокий приоритет, поскольку это может вызвать конфликты с потоками обслуживающими ввод/вывод.

В большинстве случае вместо повышения приоритета, вы можете уменьшать его. Это позволяет менее важным задачам не забирать слишком много времени от более важных задач.

Управление приоритетами потока также очень полезно на серверах и в некоторых случаях вы можете пожелать управлять приоритетом на основе логина. Если администратор подключается, то вы можете пожелать увеличить его приоритет, по сравнению с другими пользователями.

16.2.2. Инкапсуляция

Использование потоков позволяет отделить каждую задачу от других, чтобы они меньше влияли друг на друга.

Если вы использовали Windows 3.1, то вы вероятно помните, как одно плохое приложение могло легко остановить всю систему. Потоки предотвращают подобное. Без потоков, все задачи должны были реализрваться в одном учвастке кода, создавая дополнительные сложности. С потоками, каждая задача может быть разделена на независимые секции, делая ваш код проще для программирования, когда требуется одновременное выполнение задач.

16.2.3. Безопасность

Каждый поток может иметь свои собственные атрибуты безопасности, базируясь на аутентификации или других критериях. Это особенно полезно для серверных реализаций, где каждый пользователь имеет поток, ассоциированный с его соединением. Это позволяет операционной системе, реализовать должную безопасность для каждого пользователя, ограничивая его доступ к файлам и другим системным объектам. Без этого свойства вы должны бы реализовывать безопасность, возможно оставляя дыры в безопасности.

16.2.4. Несколько процессоров

Потоки могут автоматически использовать множество процессоров если они доступны . Если потоки не используются в вашем приложении, то вы имеете только один главный кодовый поток. Поток может исполняться только на одном процессоре и поэтому ваше приложение не исполняется максимально быстро.

Другие процессы могут использовать другие процессоры, так же как и операционная система. Вызовы сделанные к операционной системе вашим приложением внутренне многопоточны и ваше приложение все равно получает определенное ускорение. В дополнение, время исполнения вашего приложение может лучше, поскольку и другие процессоры задействованы для других приложений и меньше нагружают ваш процессор.

Наилучший путь получить преимущества от нескольких процессоров, это использование нескольких потоков в вашем приложении. Это не только позволяет вашему приложению использовать несколько процессоров, но и дает больше процессорного времени вашему приложению, поскольку у вас больше потоков.

16.2.5. Не нужна последовательность

Потоки предоставляют подлинное параллельное выполнение. Без потоков все запросы должны выполняться в одном потоке. Для этого работа каждой задачи должна быть разделена на малые части которые можно быстро выполнить. Если любая часть блокируется или требует много времени для исполнения, то все остальные части должны быть задержаны, пока она не выполнится. После того как одна части выполнится, начинается выполнение другой и так далее.

С потоками, каждая задача может быть закодирована отдельно и операционная система разделит процессорное время между этими частями.

16.3. Процессы против потоков

Процессы отличаются от потоков, но их часто путают. Процесс это полностью законченный экземпляр приложения, который требует ресурсов для начала выполнения, включая передачу управления операционной системе и распределение дополнительной памяти. Преимущество процессов состоит в том, что они полностью изолированы один от другого, тогда как потоки изолированы только частично. Если процесс падает, то все остальные процесс остаются в неприкосновенности.

16.4. Потоки против процессов

Потоки подобны процессам и являются параллельно выполняемым кодом. Но потоки являются частью родительского процесса. Каждый поток имеет свой собственные стек, но использует совместную куча вместе с другими потоками, того же процесса. Потоки быстрее создавать, чем процесс. Потоки также создают меньшую нагрузку на операционную систему и требуют меньше памяти для работы.

Поскольку потоки не полностью изолированы друг от друга, то их взаимодействие друг с другом значительно проще.

16.5. Переменные потоков

Переменные потока объявляются с использование ключевого слова ThreadVar.

Переменные потока подобны глобальным переменным и объявляются подобным образом. Различие в том, что обычная глобальная переменная является глобальной для всех потоков, а переменные потока специфичны для каждого потока. Так в каждом потоке, появляется свое собственное глобальное пространство.

Переменные потока могут быть полезны, когда трудно передавать ссылки на объект между библиотеками или изолированными кусками кода. Конечно переменные потока имеют и ограничения. Переменные потока не могут быть использованы или объявлены внутри пакетов. Везде где только возможно, должны быть использованы члены-переменные в классе потока. Они обеспечивают меньшую нагрузку и доступны для использования в пакетах.

16.6. Термины потоковый (*threadable*) и потоко-безопасный (*threadsafe*)

Термин потоко-безопасный (*threadsafe*) часто используется или трактуется неправильно. Его часто применяют одновременно к потоковый (*threadable*) и потоко-безопасный (*threadsafe*), что приводит к ошибкам. В данном тексте, термины потоковый и потоко-безопасный точно определены и имеют разное значение.

16.6.1. Термин потоковый (*threadable*)

Термин потоковый означает, что объект может использоваться внутри потока или использоваться потоком, если он должным образом защищен. Объект помечается как потоковый и обычно не имеет зависимости от потоков.

Если объект потоковый, то это означает, что он может использоваться в один потоке в каждый момент времени. Это может быть обеспечено созданием его локально в потоке или через глобальную защиту ресурсов.

Примеры потоковых переменных – это значения типа Integer, String и другие ординарные типы, TList, TStringList и большинство невизуальных классов.

Объект может иметь доступ к глобальным переменным или элементам управления GUI. Неограниченное (а часто и не нужное) использование глобальных переменных в большинстве случаях, это то что мешает сделать компоненты потоковыми.

Объект может быть библиотекой, компонентом, процедурой или .

16.6.2. Термин потоко-безопасный (*threadsafe*)

Термин потоко-безопасный означает, что объект осведомлен о потоках и имеет внутреннюю свою защиту ресурсов. Потоко-безопасные значения объекты могут использоваться в одном или нескольких потоках без применения защиты ресурсов.

Примером потоко-безопасных классов является VCL класс TThreadList, а также потоко-безопасные классы Indy. Конечно операционная система также потоко-безопасна.

16.7. Синхронизация

Синхронизация – это процесс передачи информации из вторичного потока основному. VCL поддерживает это с помощью метода Sychronize класса TThread.

16.8. Класс TThread

Класс TThread это класс реализации потоков, который включен в VCL и предоставляет неплохую базу для построения потоков.

Для реализация потока класс наследуется от TThread и переписывается метод Execute.

16.9. Компонент TThreadList

Компонент TThreadList – это потоко-безопасная реализация класса TList. Класс TList может быть использован в любом количестве потоков без необходимости защиты от одновременного доступа.

Класс TThreadList работает подобно TList, но не совсем также. Некоторые методы, такие как as Add, Clear и Remove аналогичны. Для других операций, класс the TThreadList должен быть заблокирован с помощью метода LockList. метод LockList – это функции и она возвращает ссылку на внутренний экземпляр класса TList. Когда он заблокирован, все другие потоки будут заблокированы. По этому, очень важно разблокировать (*Unlock*) как можно быстрее.

Пример операций с TThreadList:

```
with MyThreadList.LockList do
try
  t := Bytes div {/} KILOry
  for i := 0 to Count - 1 do
  begin
    // Operate on list items
    Items[i] := Uppercase(Items[i]);
  end;
finally
  MyThreadList.UnlockList;
end;
```

Очень важно, что бы список всегда был разблокирован по окончанию кода и поэтому всегда блокирование и разблокирование должны делаться в защитном блоке try..finally. Если список остается заблокированным, то это приведет к зависанию других потоков при попытке их доступа к списку.

16.10. Indy

Indy содержит много дополнительных классов, которые дополняют VCL возможностями поддержки потоков. Данные классы в действительности независимы от ядра Indy и полезны и для всех приложений. Они существуют в Indy, поскольку Indy разработана для работы с потоками. Indy не только использует эти классы для серверных реализаций, но и предоставляет их разработчику. Данная глава предоставляет краткий обзор этих классов.

16.11. Компонент TIdThread

Компонент TIdThread – это наследник от TThread и добавляет дополнительные расширенные свойства, большинство из которых предназначены для построения серверов и также предоставляет поддержку пулов потоков и повторного использования.

Если вы знакомы с потоками VCL, то очень важно принять во внимание, что TIdThread резко различается в нескольких ключевых областях. В TThread, метод Execute должен быть перегружен в наследниках, но в TIdThread должен быть перегружен метод Run. Ни в коем случаен не перегружайте метод Execute класса TIdThread, так как это будет препядствовать внутренним операциям TIdThread.

Для всех наследников TIdThread, метод Run должен быть перегружен. Когда поток становится активным, выполняется метод Run. Метод Run последовательно вызывается в TIdThread, пока поток не будет остановлен. Это может быть не очевидно для большинства клиентских программ, тем не менее это может быть особо полезно для всех серверов и некоторых случаев на клиенте. В этом также отличие от метода Execute класса TThread. Метод Execute вызывается только один раз. Когда завершается Execute, то поток также завершен.

Есть и другие различия между TIdThread и TThread, но они не так значительны, как Run и Execute.

16.12. Класс TIdThreadComponent

Класс TIdThreadComponent – это компонент, который позволяет вам визуально строить новые потоки, просто добавляя событие в дизайн тайм. Это основано на визуальной инкапсуляции TIdThread, что позволяет делать новые потоки очень просто.

Для использования TIdThreadComponent добавьте его на вашу форму, определите событие OnRun и установите свойство Active. Пример использования TIdThreadComponent можно увидеть в демонстрационном примере TIdThreadComponent, доступным с сайта проекта.

16.13. Метод TIdSync

Метод TThread имеет метод Synchronize, но он не имеет возможности передавать параметры в синхронизационные методы. Метод TIdSync имеет такую возможность. Метод TIdSync также позволяет возвращать значения из главного потока.

16.14. Класс TIdNotify

Синхронизация прекрасна, когда количество потоков небольшое. Тем не менее в серверных приложениях со многими потоками, они становятся узким горлышком и могут серьезно снизить производительность. Для решения этой проблемы, должны использоваться оповещения. В Indy класс TIdNotify реализует оповещения. Оповещения позволяют общаться с главным потоком, но в отличие от синхронизации потока неблокируют его, пока оповещение обрабатывается. Оповещения выполняют функцию, подобную синхронизации, но без снижения производительности.

Тем не менее, оповещения имеют и ряд ограничений. Одно из них, что значение не может быть возвращено из главного потока, поскольку оповещения не останавливают вызоваюший поток.

16.15. Класс TIdThreadSafe

Класс TIdThreadSafe – это базовый класс, для реализации потоко-безопасных классов. Класс TIdThreadSafe никогда не используется сам по себе и разработан только как базовый класс.

Indy содержит уже готовых наследников: TIdThreadSafeInteger, TIdThreadSafeCardinal, TIdThreadSafeString, TIdThreadSafeStringList, TIdThreadSafeList. Эти классы могут использоваться для потоко-безопасных версий типов integer, string и так далее. Они могут затем могут безопасно использоваться в любом количестве потоков, без необходимости заботиться об этом. В дополнение они поддерживают явное блокирование для расширенных целей.

16.16. Общие проблемы

Наибольшей проблемой при работе с потоками является параллельное выполнение. Поскольку потоки выполняются параллельно, то имеется проблема доступа к общим данным. При использовании потоков в приложении, возникают следующие проблемы:

При выполнение клиентов в потоках, приносит следующие проблемы с конкурированием:

1. обновление пользовательского интерфейса из потока.
2. общение с главным потоком из вторичных потоков.
3. доступ к данным главного потока из вторичных потоков.
4. возвращение результата из потока.
5. определение момента завершения потока.

16.17. Узкие места

Многие разработчики создают многопоточные приложения, которые работают нормально, пока количество потоков маленькое, но приложение деградирует, когда количество потоков увеличивается. Это эффект узкого горлышка. Эффект узкого горлышка проявляется когда какой то кусок кода блокирует другие потоки и другие потоки вынуждены ожидать его завершения. Не важно насколько быстр остальной код, проблема только в одном медленном куске. Код будет работать настолько быстро, насколько быстро работает его самая медленная часть.

Многие разработчики вместо поиска узкого горлышка, тратят свое время на улучшение частей кода, который они считают недостаточно быстрым, тогда как узкое горлышко не позволяет достичь какого-либо эффекта.

Обычно устранение одного узкого горлышка дает ускорение, больше чем сотни других оптимизаций. Поэтому, сфокусируйтесь на поиске узкого горлышка. И только после этого смотрите другой код для возможной оптимизации.

Несколько узких мест будет рассмотрено ниже.

16.17.1. Реализация критических секций

Критические секции эффективным и простым способом позволяют управлять доступом к ресурсам, чтобы только один поток имел доступ к ресурсу одновременно.

Часто одна критическая секция используется для защиты множества ресурсов. Допустим, что ресурсы A, B и C имеют одну общую критическую секцию для их защиты, хотя каждый ресурс независим. Проблема возникает, когда используется B, то A и C также заблокированы. Критические секции простые и выделенная критическая секция должна использоваться для каждого ресурса.

Критические секции иногда могут заблокировать слишком много кода. Количество кода, между методами Enter и Leave в критической секции должно быть минимально возможным и в

большинстве случаев должно использоваться несколько критических секций, если это возможно.

16.17.2. Класс TMREWS

Класс TMREWS может дать значительно увеличение производительности перед критическими секциями, если в основном доступ идет только по чтению и только иногда по записи. Тем не менее, класс TMREWS более сложный, чем критическая секция и требуется больше кода для установки блокировки. Для небольших кусков кода, даже если запись минимальна, обычно критическая секция работает лучше, чем TMREWS.

16.17.3. Синхронизация (*Synchronizations*)

Синхронизация обычно используется для обновления пользовательского интерфейса.

Проблема состоит в том, что синхронизация останавливает поток, пока она не будет закончена. Поскольку только главный поток обслуживает синхронизацию, только он может выполняться и остальные потоки выстраиваюися в очередь. Это делает все синхронизации самым главным узким горлышком.

Используйте оповещение везде где только возможно.

16.17.4. Обновление пользовательского интерфейса

Многопоточные приложения часто делают слишком много обновлений пользовательского интерфейса. Потеря производительности быстро наступает из-за задержек в обновлении пользовательского интерфейса. В большинстве случаев многопоточные приложения выполняются медленнее, чем однопоточная версия.

Особое внимание должно быть сделано при реализации серверов. Сервер есть сервер и основная его задача обслуживать клиентов. Пользовательский интерфейс вторичен в данном случае. Поэтому пользовательский интерфейс лучше сделать в отдельном приложении, которое будет специальным клиентом для сервера. Другое решение – это использовать оповещения или пакетные обновления. В обоих этих случаях пользовательский интерфейс немного будет заторможен, но лучше иметь пользовательский интерфейс, который будет задержан на одну секунду, чем 200 клиентов, каждый их которых будет задержан на ту же секунду. Королями являются клиенты.

17. Серверы

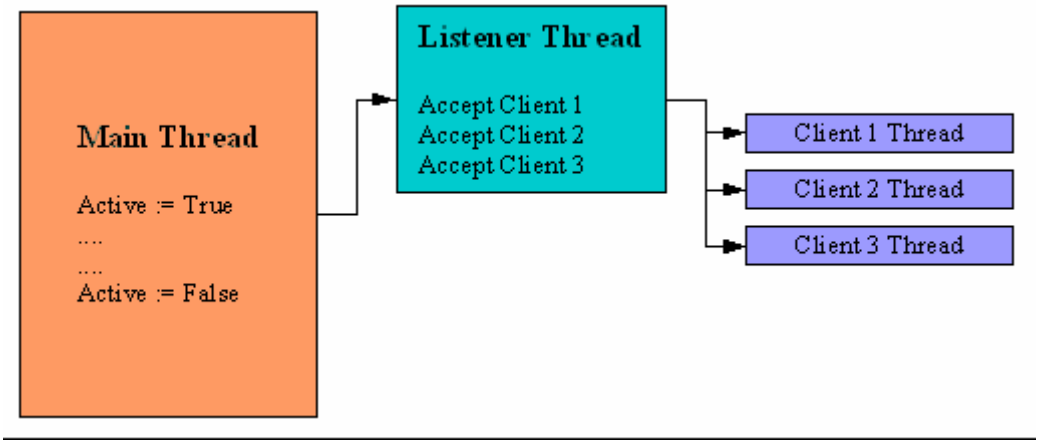
В Indy есть несколько серверных моделей, в зависимости от ваших потребностей и используемых протоколов. Следующие несколько глав введут вас в серверные компоненты Indy.

17.1. Типы серверов

17.1.1. Класс TIdTCPServer

Наиболее известный сервер в Indy – это TIdTCPServer.

Класс TIdTCPServer создает вторичный слушающий поток, который независим от главного потока программы. Слушающий поток следит за входящими запросами от клиентов. Для каждого клиента, которому он отвечает, он создает новый поток, для специфического сервиса, индивидуального соединения. Все соответствующие события затем возбуждаются к контексте этого потока.



17.1.1.1. Роль потоков

Сервера Indy разработаны вокруг потоков и работают подобно тому, как в Unix работают сервера. Приложения Unix обычно взаимодействуют со стеком напрямую с минимумом абстракции, а то и без нее. Indy изолирует программиста от стека, используя высокий уровень абстракции и реализует многие детали внутри, автоматически и скрытно.

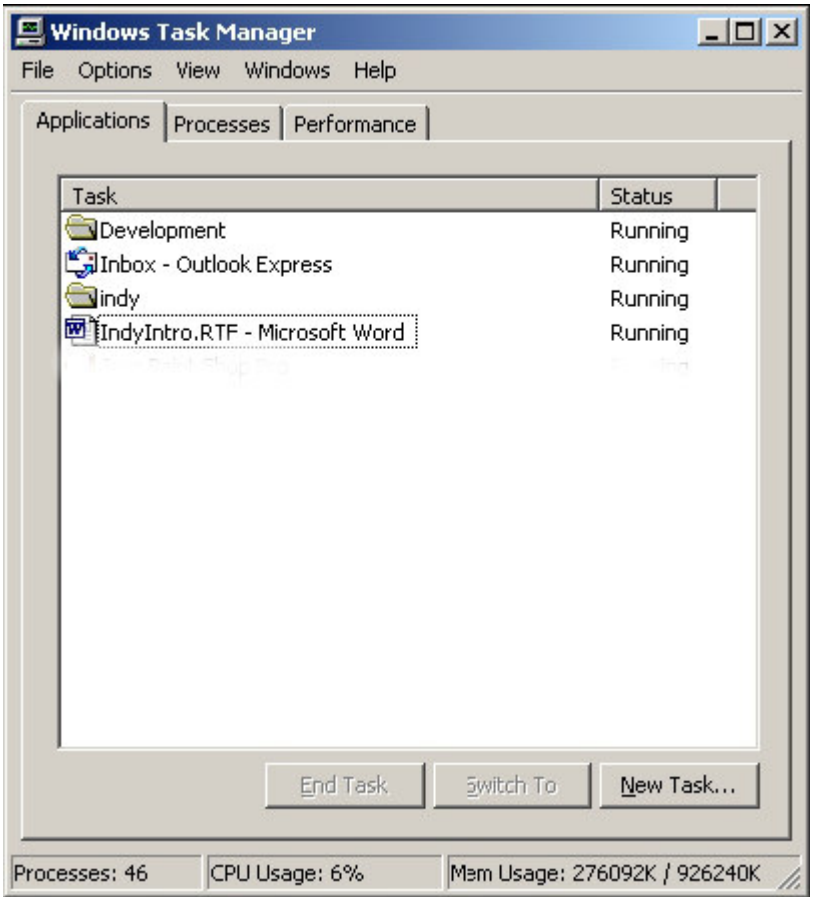
Обычно, сервера Unix имеют один или несколько слушающих процессов, которые следят за запросами от клиентов. При запросе от клиента, который сервер акцептирует, сервер размножает процесс, для обработки каждого клиентского соединения. Обслуживание множества клиентов таким образом очень просто, так как каждый процесс работает только с одним клиентом. Процесс также может запускаться в своем контексте безопасности, который может быть установлен слушателем или процессом на основе аутентификации, прав или других основаниях.

Сервера Indy работают подобным образом. Windows в отличии от Unix не размножает процессы, но зато Windows хорошо работает с потоками. Сервера Indy размещают поток для каждого клиентского соединения, вместо создания отдельного законченного процесса, как это делает Unix. Это дает все преимущества процессов, без наличия недостатков.

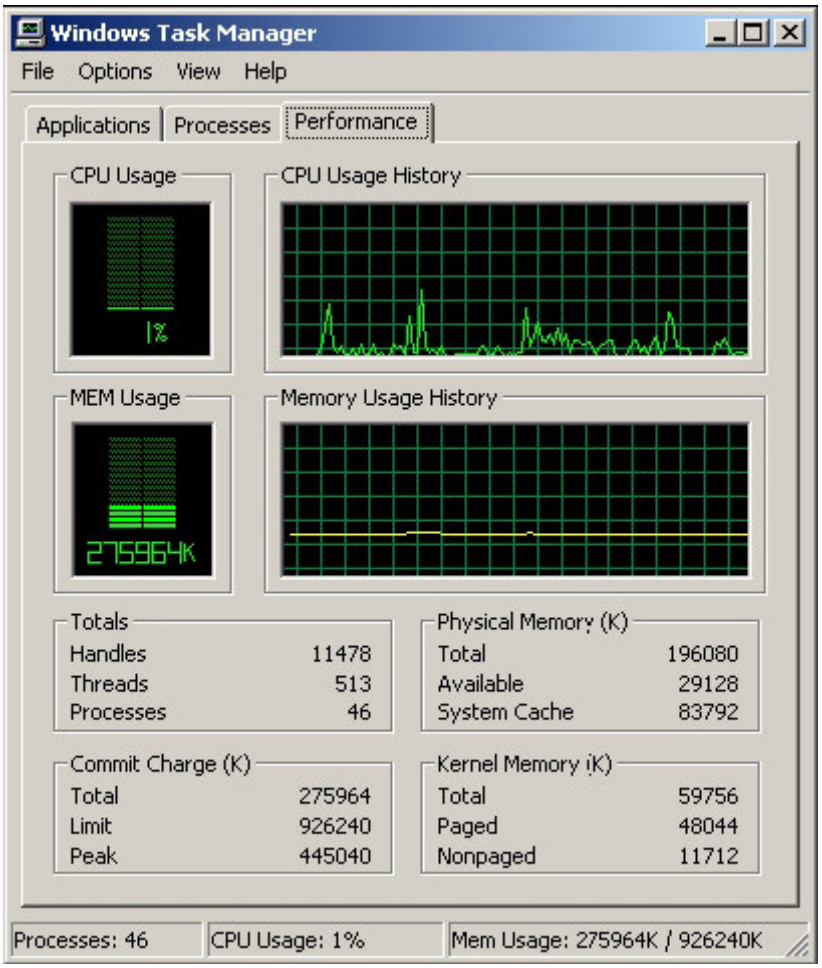
17.1.1.2. Сотни потоков

Для сильно нагруженного сервера может потребоваться сотни или даже тысячи потоков. Есть общее убеждение, что сотни и тысячи потоков могут убить вашу систему. Это неверное убеждение.

Количество потоков, которые запущены на вашей системе, может удивить вас. При минимальном количестве запущенных серверов и указанными запущенными приложениями:



Моя система имеет 513 созданных потоков:



Вы видите, что даже при 513 потоках процессор нагружен только на 1%. Сильно нагруженный сервер IIS (Microsoft Internet Information Server) может создать сотни и тысячи потоков. Данный тест был выполнен на Pentium III 750 MHz с 256 MB оперативной памяти.

На большинстве серверов, потоки находятся в режиме ожидания данных. При ожидании на блокирующем сокете, поток находится в неактивном состоянии. Поэтому на сервере из 500 потоков, только 25 могут оказаться активными одновременно.

В сокетных приложениях, наибольшие ограничения накладывает самый медленный компонент. В сетевых приложениях, сетевая плата обычно самый медленный компонент и поэтому ничего не сможет превысить возможностей сетевой платы. Даже самая быстрая сетевая плата, во много раз медленнее процессора, и является узким местом, если она задействована по полной.

17.1.1.3. Реальные ограничения на потоки

Реально средняя система начинает испытать проблемы, только когда процесс создаст свыше 1000 потоков, из-за нехватки памяти. Размер стека потока может быть уменьшен за счет увеличения количества потоков, тем не менее, надо рассмотреть и другие альтернативные причины.

Большинству серверов требуется только несколько сотен потоков. Тем не менее, сильно нагруженные сервера или сервера, у которых трафик низкий, но множество подключенных

клиентов, таких как чат (*Chat*) сервер требуют другой реализации. Такие сервера, в Indy могут создавать тысячи потоков.

Также следует понять, что количество клиентов совсем не означает тоже количество конкурирующих потоков. Когда каждый клиент выделяет отдельный поток, то поток выделяется клиенту только при соединении. Многие сервера обслуживают кратко живущие соединения, такие как HTTP сервера. HTTP соединения для страниц обычно живут только одну секунду или менее, особенно если используется прокси или кэширование. Предположим 1 секунду на соединение и только 500 потоков, что позволяет до 30 000 клиентов в час.

Indy 10 имеет и другие модели, в дополнение к потокам серверов. Возможности Indy 10 ограничены только размером доступной памяти для размещения сокетов.

17.1.1.4. Модели серверов

Есть два пути построения TCP серверов: с командными обработчиками и с событиями OnExecute. Командные обработчики делают построение серверов много проще, но не во всех ситуациях.

Командные обработчики удобны для протоколов, которые обмениваются командами в текстовом формате, но не очень удобны для протоколов, которые имеют команды двоичной структуры или совсем не имеют командной структуры. Большинство протоколов текстовые и могут использоваться командные обработчики. Командные обработчики полностью опциональны. Если они не используются сервера Indy продолжают использовать старые методы. Командные обработчики рассмотрены в деталях в главе «Командные обработчики».

Некоторые протоколы являются двоичными или не имеют командной структуры и не пригодны для использования командных обработчиков. Для таких серверов должно использоваться событие OnExecute. Событие OnExecute постоянно вызывается пока существует соединение и передает соединение, как аргумент. Реализация очень простого сервера с использованием события OnExecute выглядит так:

```
procedure TformMain.IdTCPServer1Execute(AThread: TIdPeerThread);
var
  LCmd: string;
begin
  with AThread.Connection do
  begin
    LCmd := Trim(ReadLn);
    if SameText(LCmd, 'QUIT') then
    begin
      WriteLn('200 Good bye');
      Disconnect;
    end
    else if SameText(LCmd, 'DATE') then
    begin
      WriteLn('200 ' + DateToStr(Date));
    end
    else
    begin
      WriteLn('400 Unknown command');
    end;
  end;
end;
```

здесь нет необходимости проверять действительность соединения, так как Indy делает это автоматически. Так же нет необходимости производить опрос, поскольку и это Indy делает

автоматически за вас. Она вызывает событие периодически, пока соединение не прекратится. Это может быть вызвано или явным отсоединением, или по сетевой ошибке, или если клиент отсоединился. В действительности, не требуется делать никаких опросов на предмет отсоединений. Если данный опрос все же необходимо делать, вам надо только позаботиться об возбуждении исключений, что бы Indy смог нормально их отработать.

17.1.1.5. Командные обработчики

Indy 9.0 содержит новое средство, называемое командные обработчики. Командные обработчики – это новая концепция, используемая в TIdTCPServer, которая позволяет серверу выполнять парсинг команды и обрабатывать его для вас. Командные обработчики это разновидность «визуального управления сервером» и является только маленьким заглядыванием в будущие реализации Indy.

Для каждой команды, которую вы желаете обрабатывать сервером, создается командный обработчик. Думайте об командных обработчиках, как о списке действий для сервера. Командный обработчик содержит свойства, которые указывают, как парсить параметры, команды, некоторые действия, которые могут быть выполнены автоматически и дополнительные автоматические ответы. В некоторых случаях используя только свойства, вы сможете создать полностью функциональную команду без необходимости написания какого либо кода. Каждый командный обработчик имеет уникальное событие OnCommand. Когда событие вызывается, то нет необходимости определять какая команда была запрошена, так как данное событие уникально для каждого командного обработчика. В дополнение, командный обработчик уже распарсил параметры и выполнил автоматические действия для вас.

Вот маленький пример использования командных обработчиков. Во-первых вы должны определить две команды: QUIT и DATE. Создадим два командных обработчика, как указано ниже:

Для cmdhQuit свойство disconnect устанавливается в true. Для cmdhDate событие OnCommand определяется так:

```
procedure TForm1.IdTCPServer1cmdhDateCommand(ASender: TIdCommand);
begin
  ASender.Reply.Text.Text := DateTimeToStr(Date);
end;
```

это законченный код командного обработчика. Все другие детали, специфицированы установкой свойств в командных обработчиках.

Командные обработчики более подробно описаны в главе «командные обработчики».

17.1.2. Класс TIdUDPServer

Поскольку UDP не требуется соединения (по определению), то класс TIdUDPServer работает отлично от TIdTCPServer. Подобно TIdSimpleServer класс TIdUDPServer не имеет некоторых режимов, и поскольку UDP не требуется соединения, TIdUDPClient имеет только слушающие методы.

При активизации, класс TIdUDPServer создает слушающий поток, для прослушивания входящих UDP пакетов. Для каждого, принятого UDP пакета, класс TIdUDPServer возбуждает событие OnUDPRead в главном потоке или в контексте слушающего потока, в зависимости от значения свойства ThreadedEvent.

Когда значение свойства ThreadedEvent = false, то событие OnUDPRead возбуждается в контексте главного потока программы. Когда значение свойства ThreadedEvent = true, то событие OnUDPRead возбуждается в контексте слушающего потока.

Когда значение свойства ThreadedEvent равно false, то блокируется прием дальнейших сообщений. Поэтому событие OnUDPRead должно быть как можно более быстрым.

17.1.3. Класс TIdSimpleServer

Класс TIdSimpleServer предназначен для разового использования серверов. Класс TIdSimpleServer предназначен для обслуживания одного соединения за раз. Хотя он может обслуживать другие запросы по окончанию, обычно он используется только для одного запроса.

Класс TIdSimpleServer не создает потоков для прослушивания или вторичных потоков соединения. Вся функциональность реализована в одном потоке.

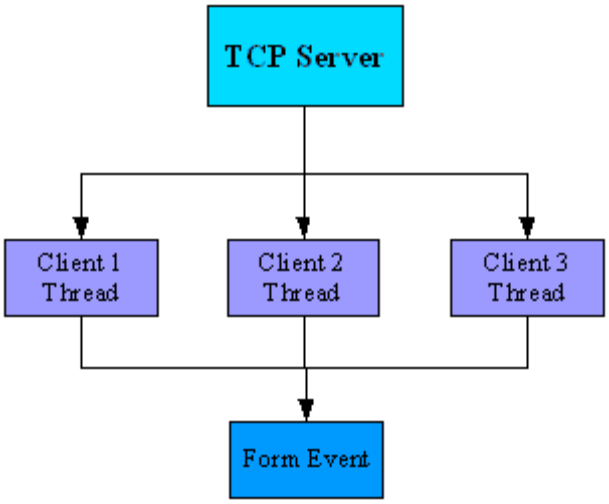
Компонент клиента TIdFTP использует класс TIdSimpleServer. Когда FTP выполняет передачу, вторичное TCP соединение открывается для передачи данных и закрывается, когда данные будут переданы. Данное соединение называется «канал данных (data channel)» и оно уникально для каждого передаваемого файла.

17.2. События потоков

События TIdTCPServer потоковые. Это означает, что они не являются частью потока, они выполняются внутри потока. Это очень важная деталь. Будьте уверены, что вы понимаете эту деталь до начала обработки.

Это сначала может показаться странным, что событие может быть частью формы но при этом выполняться внутри потока. Тем не менее, это было умышленно так сконструировано, чтобы события созданные в дизайн тайм, были подобны любым другим событиям, без необходимости создания пользовательского класса и перекрытия метода.

В создаваемых компонентах наследниках, перекрытие тоже доступно. Но для построения приложений, модель обработчиков событий намного проще в применении.



Каждый клиент создает свой собственный поток. При использовании данного потока, события TCP сервера (который является частью формы или модуля данных) вызываются из данных потоков. Это означает, что одиночное событие может быть вызвано множество раз из разных

потоков. Такие события получают в качестве аргумента AThread, который указывает на поток из которого возбуждено событие.

Примерами потоковых события являются командные обработчики сервера OnConnect, OnExecute и OnDisconnect.

17.3. Модели TCP серверов

TCP сервер Indy поддерживает две модели для построения серверов. Данные методы это OnExecute и командные обработчики. Indy 8 поддерживает только OnExecute.

17.3.1 Событие OnExecute

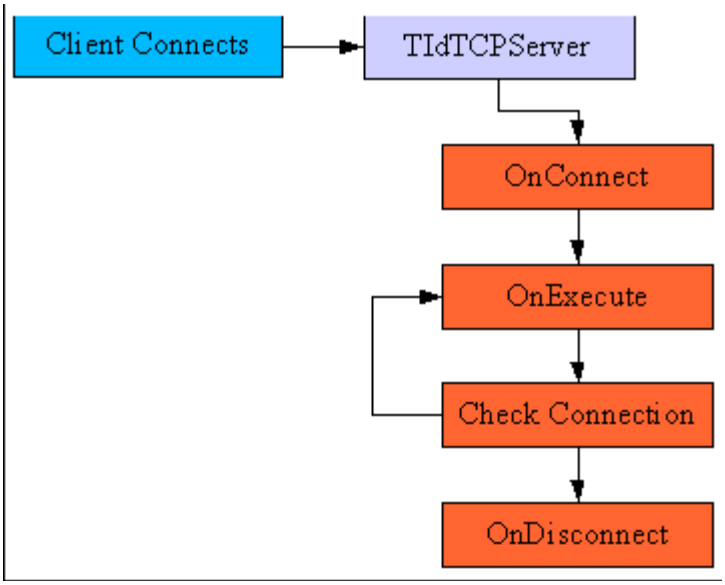
Событие OnExecute ссылается на событие OnExecute класса TIdTCPServer. При реализации сервера по данной модели, должно быть определено событие OnExecute или перекрыт метод DoExecute.

Модель OnExecute допускает полный контроль разработчиком и позволяет реализовывать любые типы протоколов, включая двоичные протоколы.

После подсоединения клиента к серверу, возбуждается событие OnExecute. Если событие OnExecute не определено, то возбуждается исключение. Событие OnExecute возбуждается в цикле, как только подсоединяется клиент. Это очень важная деталь и поэтому разработчик должен побеспокоиться об

1. Помнить о том, что событие возникает в цикле.
2. Не препядствовать Indy выполянть обработку в цикле.

Внутренний цикл показан на следующей диаграмме:



На этапе проверки соединения выполняется следующие проверки:

- Клиент еще подсоединен
- Disconnect не был вызван во время OnExecute
- Отсутствуют фатальные ошибки
- Не было возбуждено необработанное исключение в OnExecute

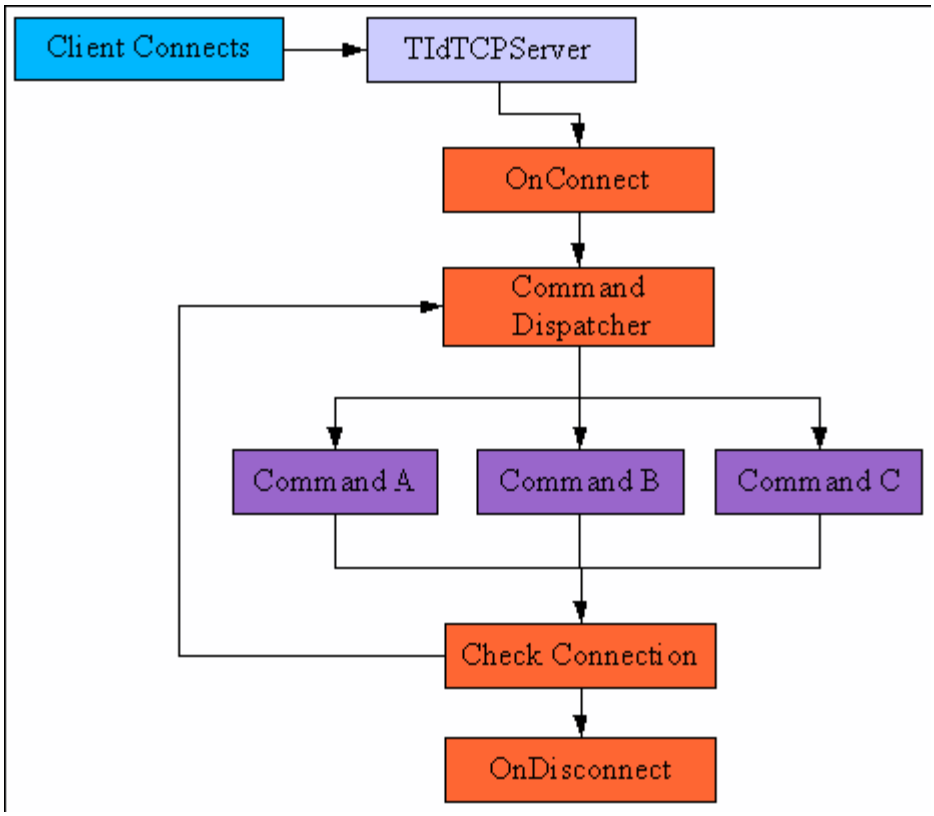
- Сервер еще активен

Если все эти проверки и другие проверки истинны, то событие OnExecute возбуждается снова. Поэтому, разработчик никогда не должен конструировать свой цикл внутри OnExecute, который будет дублировать это, так как это будет мешать Indy.

17.3.2. Обработчики команд (*Command Handlers*)

Командные обработчики подобны спискам действий для построения серверов, в стиле визуальной среды проектирования. Командные обработчики ограничены текстовыми протоколами. Данные, передаваемые по протоколу, могут быть двоичными.

Обработчики команд автоматически читают и разбирают команды. При этом возбуждается специфическое событие OnCommand.



17.4. Обработчики команд (*Command Handlers*)

Создание серверов в Indy всегда было достаточно простой задачей, тем не менее в Indy 9 это стало проще после введения командных обработчиков в класс TCP сервера (TIdTCPServer).

Обработчики команд подобны спискам действий для сервера. Командные обработчики работают следующим образом: - вы создаете обработчик команд для каждой команды и затем используя обработчик команд определяете поведение для конкретной команды. Когда команда принята от клиента, то сервер автоматически разбирает ее и передает конкретному обработчику. Обработчики команд не только имеют свойства для настройки поведения, но также методы и события.

Обработчики команд работают только с текстовыми командами и ответами TCP протоколов. Тем не менее это покрывает нужды почти 95% серверов используемых в настоящее время. Хотя обработчик в состоянии работать с двоичными данным, но сами команды могут быть только

текстовыми. Имеется некоторое количество протоколов, которые работают с помощью двоичных команд. Для протоколов, использующих двоичные команды или текстовые команды, которые не совместимы (от корректора: труднореализуемые??? Не могу себе представить что он имел в виду, потому что conversational это словоохотливый, разговорчивый), реализация обработчиков команд не является обязательной что позволяет сохранить обратную совместимость .

17.4.1. Реализация

Класс TCPServer содержит свойство, именуемое CommandHandlers, которое является коллекцией обработчиков команд. Обработчики обычно создаются во время разработки, тем не менее, при реализации наследников они могут создаваться и во время исполнения. Если командные обработчики создаются во время исполнения, то они должны быть созданы путем перекрытия метода InitializeCommandHandlers. Это гарантирует, что они создадутся только во время исполнения. Если они создаются в конструкторе, они будут создаваться каждый раз, когда TCPServer загружается из потока и записывается обратно в поток. Это может привести к созданию множества копий для каждого обработчика. Инициализация, напротив, вызывается только однажды после первой активации TCPServer.

Класс TCPServer содержит несколько свойств и событий имеющих отношение к обработчикам команд. Свойство CommandHandlersEnabled разрешает или запрещает работу обработчика как единого целого. Свойство OnAfterCommandHandler возбуждается после выполнения каждого обработчика и событие OnBeforeCommand возбуждается перед выполнением каждого обработчика. Событие OnNoCommandHandler возбуждается если обработчик, соответствующий команде, не найден.

Если свойство CommandHandlersEnabled равно true и определены обработчики, то выполняется их обработка. Иначе вызывается событие OnExecute, если оно назначено. Обработчик OnExecute не вызывается если были обработаны команды.

Если есть соединение, TCPServer читает строки текста из соединения и пытается найти подходящий обработчик команд. Любые пустые строки игнорируются. Для непустых строк сначала возбуждается событие OnBeforeCommandHandler. Затем ищется подходящий командный обработчик. Если командный обработчик найден и его свойство enabled установлено в true, то возбуждается его событие OnCommand, а иначе возбуждается событие OnNoCommandHandler. После всего этого возбуждается событие OnAfterCommand.

17.4.2. Пример протокола

Для демонстрации базовой реализации обработчиков команд, определим простой протокол. Для демонстрации пользовательского сервера времени реализуем три команды:

- **Help** – показывает список поддерживаемых команд и их форматы.
- **DateTime <format>** - возвращает текущую дату и время в указанном формате, если формат не указан, то по умолчанию используется формат yyyy-mm-dd hh:nn:ss.
- **Quit** – закрывает сессию и отсоединяется.

Это очень простая базовая реализация, но она работает вполне приемлемо для целей демонстрации. Конечно, вы можете ее расширить для того чтобы изучить возможности обработчиков команд.

17.4.3. Базовый пример

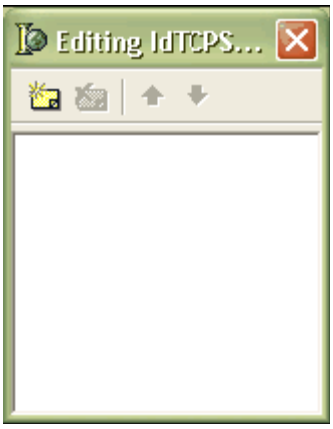
Сначала сконструируем базу для построения примера. Это подразумевает, что вы уже умеете работать с TIdTCPServer и вот ваши следующие шаги. Для построения базового примера выполним следующие шаги:

1. Создадим новое приложение.
2. Добавим TIdTCPServer на форму.
3. Установим свойство TIdTCPServer.Default в 6000. порт 6000 это порт для демо, выбранный случайным образом, можно использовать любой свободный порт.
4. Установим свойство TIdTCPServer.Active в True. Это должно активировать сервер при старте приложения.

Этим мы создали базовое приложение. Оно пока ничего не делает, поскольку еще нет ни обработчиков команд, ни событий.

17.4.4. Создание обработчика команд

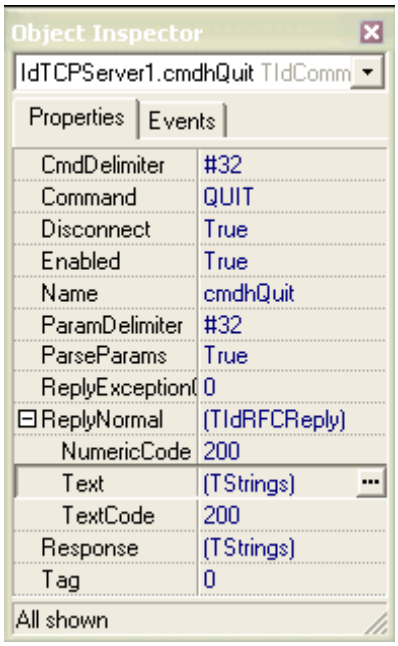
Обработчики команд создаются при редактировании свойства CommandHandlers класса TIdTCPServer. Свойство CommandHandlers – это коллекция. Обработчики могут быть модифицированы как во время исполнения, так и во время разработки. Для редактирования обработчиков во время разработки нажмите на кнопку <…> на свойстве CommandHandlers в инспекторе объектов. Появится следующий диалог:



Он пока пуст, поскольку еще нет ни одного обработчика. Для создания обработчика команд, или нажмите правую кнопку мыши и выберите пункт Add, или нажмите первую кнопку в панели инструментов диалога. После это в списке появится обработчик команд.

Для редактирования обработчика выберите его в инспекторе объектов. Редактирование обработчиков подобно редактированию полей набора данных БД или колонок в DBGrid. Если инспектор объектов не виден, то нажмите F11 для его отображения.

Инспектор объектов выглядит как на приведенном рисунке. Показано, что есть уже одно измененное свойство, реализующее команду. Это команда QUIT и она будет обсуждена ниже.



Пошаговое описание реализации команды QUIT:

1. Command = Quit – это команда сервера которую сервер будет использовать для поиска обработчика при чтении ввода. Команда не чувствительна к регистру.
2. Disconnect = True – это значит, что сервер отсоединится от клиента после получении и обработки данной команды.
3. Name = cmdhQuit – данное свойство не оказывает никакого влияния на обработку, но оно предназначено для упрощения идентификации обработчика в коде. Данный шаг необязательный.
4. ReplyNormal.NumericCode = 200 – Команды обычно возвращают 3-х разрядный код и необязательный текст. Задав это свойство, мы указывем обработчику возвращать в ответ на команду код 200 и дополнительный текст из ReplyNormal.Text если конечно не произойдет ошибка во время выполнения команды.
5. ReplyNormal.Text = Good Bye – дополнительный текст, который посылается вместе с ReplyNormal.NumericCode.

После этого мы имеем полностью работоспособный обработчик команд.

17.4.5. Поддержка обработчика команд

Теперь когда обработчик создан, есть еще несколько глобальных параметров, относящихся к северам на текстовых командах серверов и обработчиков команд, которые также должны быть установлены. Все это свойства TIdTCPServer, а не обработчикам команд.

17.4.5.1. Свойство Greeting (*приветствие*)

Обычной практикой для серверов, является предоставления информации, приветствия от сервера, перед тем как сервер начнет обрабатывать команды клиента. Типичный ответ сервера, показывающий, что сервер готов, это код 200 и установка кода не равным нулю, разрешит посылку приветствия

Установите свойства **Greeting. NumericCode** = 200 и **Greeting.Text** в "Hello".

17.4.5.2. Свойство ReplyExceptionCode

Если во время обработки команд обнаружатся не обслуженные исключения, то используется данное свойство со значением отличным от нуля. Код 500 это типичный ответ для внутренних, неизвестных ошибок. Вместе с кодом отсылается и текстовый отзыв.

Установите ReplyExceptionCode в 500.

17.4.5.3. Свойство ReplyUnknownCommand

Если во время обработки команд обнаружатся не обработанные исключения, то данное свойство будет использовано для построения ответа, если его значение отличается от нуля. Код 400 наиболее общий ответ для подобных случаев.

Установите ReplyUnknown.NumericCode в 400 и ReplyUnknown.Text в "Unknown Command".

17.4.5.4. Прочие свойства

У TIdTCPServer есть еще и другие свойства и события, для реализации дополнительного поведения, относящего к командным обработчикам, но приведенные выше являются минимумом, который должен быть реализован.

17.4.6. Тестирование новой команды

Теперь, когда команда уже реализована можно приступить и к тестированию, просто воспользуемся Telnet, поскольку протокол текстовый:

1. Запустим приложение.
2. В меню Start: Run введем: *telnet 127.0.0.1 6000* и нажмем OK. Это указывает Telnet подсоединиться к компьютеру на порт 6000, который используется в демонстрационном примере.
3. Сервер должен ответить *200 Hello,* что является приветствием, из свойства Greeting of TIdTCPServer.
4. Telnet затем покажет каретку. Это означает, что сервер готов и ожидает команду.
5. Введем *HELP* и нажмем enter. Сервер ответит "400 Unknown Command". Поскольку пока мы не создали командного обработчика для команды HELP и ответ "400 Unknown Command" был взят из свойства ReplyUnknown.
6. Введем *QUIT*. Сервер ответит "200 Good Bye" и отсоединится от клиента.

Поздравляем! Вы построили сервер с обработчиком команд. В следующей главе мы реализуем остальные две команды - HELP и DATETIME, которые имеют отличное от команды QUIT поведение.

17.4.7. Реализация HELP

Команда HELP подобно по поведению на команду QUIT за исключением двух различий.

1. Не происходит разъединение сеанса.
2. В дополнение ответ также предоставляет текстовый отклик со справочной информацией.

Для реализации команды HELP выполним следующие шаги:

1. Создадим новый командный обработчик.
2. Command = Help
3. Name = cmdhHelp
4. ReplyNormal.NumericCode = 200
5. ReplyNormal.Text = Help Follows

Все эти шаги знакомы вам по реализации команды QUIT. Дополнительное свойство, которое здесь используется - это свойство Response, которое является списком строк. Если свойство Response содержит текст, то оно посылается клиенту после отсылки ReplyNormal. Для реализации команды HELP используется редактор строк свойства Response:

```
Help – Display a list of supported commands and basic help on each.
DateTime <format> – Return the current date and/or time using the specified
format.
If no format is specified the format yyyy-mm-dd hh:nn:ss will be used.
Quit – Terminate the session and disconnect.
```

Теперь если вы подсоединитесь к серверу и пошлете команду HELP, то сервер ответит следующим образом:

```
200 Hello
help
200 Help Follows
Help – Display a list of supported commands and basic help on each.
DateTime <format> – Return the current date and/or time using the specified
format.
If no format is specified the format yyyy-mm-dd hh:nn:ss will be used.
Quit – Terminate the session and disconnect.
.
```

17.4.8. Реализация DATETIME

Команда DATETIME – это последняя команда данного протокола. Оно отличатся и от QUIT и от HELP, в том, что требует особой функциональности, которая не может быть создана только с помощью свойств. Для реализации команды DATETIME будет использован обработчик события.

Для начала построим базовый обработчик команды, используя шаги с которым вы уже знакомы:

1. Создадим новый командный обработчик.
2. Command = DateTime
3. Name = cmdhDateTime
4. ReplyNormal.NumericCode = 200

В данный момент свойство ReplyNormal.Text не определяется, обработчик события будет его определять для каждого ответа. Для определения обработчика, используйте инспектор объектов, выбрав командный обработчик для DATETIME. Переключитесь на закладку events и создайте событие OnCommand. Delphi создаст обработчик следующим образом:

```
procedure TForm1.IdTCPServer1TIdCommandHandler2Command(ASender: TIdCommand);
begin
end;
```

В обработчик OnCommand передается аргумент of ASender типа TIdCommand. Это не командный обработчик, а сама команда. Командные обработчики глобальны для всех соединений, тогда как, команды специфичны для соединения connection и обрабатываются в рамках экземпляра события OnCommand. Это гарантирует, что обработчик выполнит корректную обработку для каждого клиентского соединения.

Перед вызовом обработчика события, Indy создает экземпляр команды и инициализирует его свойства на основе данных обработчика команд. Вы можете использовать команды для смены свойства со значений по умолчанию, вызывать методы для выполнения задач или для доступа к свойству Connection для общения с соединением напрямую.

Данный протокол определяет команду DATETIME, как имеющую дополнительный параметр, указывающий формат даты и времени. Команда (TIdCommand) реализует это с помощью свойства Params, которое является списком строк. Когда команда принимается от клиента и свойство ParseParams установлено в true (по умолчанию) Indy использует свойство CmdDelimeter (по умолчанию равное #32 или пробел) для разделения команды и параметров.

Например, в данном протоколе, клиент может послать следующее:

```
DATETIME hhnnss
```

В этом случае, свойство ASender.Params будет содержать строку "hhnnss" в свойстве ASender.Params[0]. Количество параметров может быть определено с помощью свойства ASender.Params.Count.

Используя данные свойства обработчик OnCommand может быть реализован следующим образом:

```
procedure TForm1.IdTCPServer1TIdCommandHandler2Command(ASender: TIdCommand);
var
  LFormat: string;
begin
  if ASender.Params.Count = 0 then
  begin
    LFormat := 'yyyy-mm-dd hh:nn:ss';
  end
  else
  begin
    LFormat := ASender.Params[0];
  end;
  ASender.Reply.Text.Text := FormatDateTime(LFormat, Now);
end;
```

данная реализация просто читает параметры и использует ASender.Reply.Text для посылки ответа обратно клиенту. Не требуется устанавливать ASender.Reply.NumericCode, так как Indy инициализирует его значением 200, из командного обработчика ReplyNormal.NumericCode.

Примечание: используйте свойство ASender.Reply.Text.Text. Указание слова Text дважды требуется потому что свойство Text команды это список строк и мы имеем также TStrings.Text в дополнение к этому. Поскольку это список строк, другие методы или свойства, такие как Add, Delete и другие также могут использоваться. Свойство Text используется как ASender.Reply.Text, в некоторых случаях может быть предварительно проинициализировано и запись в него вызовет перекрытия текста.

Если снова протестировать пример с помощью telnet, то теперь ответ будет таким:

```
200 Hello
datetime
200 2002-08-26 18:48:06
```

В некоторых случаях свойство Params не может быть использовано. Свойство DATETIME одно из них. Представим себе следующую команду:

```
DATETIME mm dd yy
```

В данном случае значение свойства Params.Count будет равно 3 и событие будет неверно обработано, возвратит только значение месяца (mm). Для данных случаев, когда значение параметра включает разделители, можно использовать свойство UnparsedParams. Дополнительно, свойство ParseParams можно установить в False.

Свойство UnparsedParams содержит данные независимо от свойства ParseParams, но установка ParseParams в false увеличивает эффективность, сообщая Indy, что не требуется разбирать параметры в свойство Params.

Обработчик события, модифицированный для использования с UnparsedParams:

```
procedure TForm1.IdTCPServer1TIdCommandHandler2Command(ASender: TIdCommand);
var
  LFormat: string;
begin
  if ASender.Params.Count = 0 then
  begin
    LFormat := 'yyyy-mm-dd hh:nn:ss';
  end
  else
  begin
    LFormat := ASender.UnparsedParams;
  end;
 ASender.Reply.Text.Text := FormatDateTime(LFormat, Now);
end;
```

17.4.9. Заключение

Обработчики команд очень гибки и содержат большее количество свойств и методов, чем приведено. Это только введение в обработчики команд и их возможности. Надеюсь, что этого достаточно, что вызвать у вас интерес и начать работать.

Имеется также особые планы для будущих версий – сделать командные обработчики еще более наглядными и удобными на стадии разработки.

17.5. Postal Code Server - реализация OnExecute

OnExecute Implementation

Demo

Threads

17.6. Postal Code Server – командные обработчики

Command Handlers

Greeting

CommandHandlers

ReplyException

ReplyTexts

ReplyUnknownCommand

Demo

17.7. Управление потоками

Управление потоками абстрагировано в Indy в менеджеры потоков. Менеджеры потоков позволяют иметь различные (даже пользовательские) реализации стратегий управления потоками.

Управление потоками - это необязательная дополнительная возможность. Если вы не определяете менеджер потоков в свойстве ThreadManager в компоненте, который поддерживает управление потоками (таком как TIdTCPServer) Indy неявно создает и уничтожает экземпляр менеджера потоков.

17.7.1. Класс TIdThreadMgrDefault

Стратегия управления потоками по умодчанию в Indy очень простая. Каждый раз, когда требуется поток, он создается. Когда поток не нужен, он уничтожается. Для большинства серверов – это приемлемо и пока вам не понадобится пул потоков, вы должны использовать стратегию управления потоками по умолчанию. В большинстве серверов различие в производительности ничтожное или полностью отсутствует.

Стратегия по умолчанию также дает дополнительное преимущество, так как, каждый поток гарантировано «чистый». Потоки часто распределяют память или другие объекты. Эти объекты обычно освобождаются автоматически, когда поток разрушается. Использование управления потоками по умолчанию дает вам уверенность, что вся память освобождена и все очищено. Когда используется пуле потоков, то вы должны быть уверены, что все очищено перед повторным использованием потока. Несоблюдение этого правила, может привести к тому что, пользователь будет иметь доступ к информации предыдузего пользователя.. Такие предосторожности не требуются при использовании менеджера потоков по умолчанию, поскольку все ассоциированные данные уничтожаются вместе с потоком.

17.7.2. Пул потоков (*Thread Pooling*)

Обычно диспетчеризация потоков по умолчанию вполне приемлема. Тем не менее, для серверов, которые обслуживают коротко живущие соединения, создание и уничтожение потоков, сравнимо со временем обслуживания запроса. В данной ситуации, лучше использовать пул потоков.

В пуле потоки создаются предварительно и используются повторно. Они создаются до использования и хранятся неактивными в пуле. Когда требуется поток, то он берется из пула и активируется. Если требуется больше потоков, чем есть в пуле, то создаются дополнительные потоки. Когда поток больше не требуется, то вместо его разрушения он деактивируется и возвращается в пул.

Создание и разрушение потоков может быть очень интенсивным. Это особо относится к серверам, которые обслуживают коротко живущие соединения. Такие сервера создают поток, который используется только короткое время и затем уничтожается. Это приводит к очень высокой частоте создания и уничтожения потоков. Примером таких серверов могуь служить сервера времени или даже web сервера. Посылается простой запрос и отсылается простой ответ.

При использование браузера для просмотра некоторых сайтов могут создаваться сотни соединений к серверу.

Пул потоков может смягчить данную ситуацию. Вместо создания и уничтожения потоков по требованию, потоки выдаются из списка неактивных потоков, которые уже созданы. Когда поток больше не нужен, он возвращается обратно в пул. Пока потоки находятся в пуле – они отмечены как неиспользуемые и поэтому не требуют ресурсов процессора. Как дальнейшее усовершенствование - размер пула можно настраивать динамически, в зависимости от потребностей системы. Indy имеет поддержку пула потоков. Пул потоков в Indy доступен через использование компонента TIdThreadMgrPool.

18. SSL – безопасные сокеты

SSL – это сокращение от Secure Socket Layer и является проверенным методом шифрования данных передаваемых через Интернет. SSL обычно используется для HTTP (Web) трафика и называется защищенный (secured) HTTPS. Конечно, SSL не ограничен HTTP и может быть использован с любым TCP/IP протоколом.

Для использования SSL в Indy, вы во-первых должны установить поддержку SSL. Indy реализует поддержку SSL в открытом для расширения стиле, но единственная поддерживаемая сейчас реализация библиотеки SSL - это OpenSSL. Библиотека OpenSSL доступна, как набор DLL и доступна для загрузки отдельно от дистрибутива Indy.

Экспорт некоторых методов шифрования, таких как SSL, запрещен благодаря неописуемой мудрости и понимания технологий правительством США и других стран. По этому SSL технология не может быть размещена на web сайте, без принятия определенных мер по точному определению местонахождения каждого клиента, желающего загрузить технологии. Это не только трудно для практической реализации, но и накладывает на влядельцев сайтов дополнительную ответственность.

Ограничение касается только на распространение в электронном виде, но не на предоставление исходного кода в печатном виде. Даное ограничение касается только экспорта и не является важным.

Поэтому программисты просто распечатали исходный код на футболках, пересекли границу, а затем ввели и скомпилировали его. После того как это произошло, страны, которые не подписали соглашение с США смогли свободно распространять технологию шифрования в любой форме и снять импорные ограничения для любого, кто пожелает загрузить технологию шифрования в форме пригодной для использования.

Многие проблемы были разрешены с тех пор, и некоторые правительства даже поумнели. Тем не менее, многие экспортные ограничения продолжают иметь место и различаются от страны к стране. Поэтому, все работы по Indy SSL были сделаны в Словении и технологии шифрования, относящие к Indy были также распространены через Словению. Словения не имеет ограничений на экспорт технологий шифрования.

В дополнение к экспорту технологий шифрования, некоторые страны имеют ограничения на использование и даже на владение технологиями шифрования. Вы должны проверить законодательство в вашей стране прежде чем делать реализацию с использованием SSL. Такие страны как Китай, Ирак и другие имеют суровые наказания и даже за владение такими технологиями.

18.1. Безопасные протоколы HTTP и HTTPS

Реализация протокола HTTPS в Indy очень проста. Просто укажите безопасный URL вместо стандартного URL и Indy клиент HTTP (*TIdHTTP*) все остальное сделает автоматически. Чтобы сделать URL безопасным, просто измените http:// на https://.

Примечание: чтобы HTTPS сессия была установлена, web сервер, который отвечает должен поддерживать SSL и иметь установленный сертификат шифрования. Также HTTP клиенты Indy не проверяют сертификат сервера – это ваша обязанность.

18.2. Другие клиенты

SSL может легко реализована в любом TCP клиенте Indy с помощью использования SSL IOHandler. Обычно шифрование должно быть реализовано с помощью Intercept вместо IOHandler. SSL реализация в Indy использует IOHandler вместо Intercept, поскольку SSL библиотека выполняют весь обмен самостоятельно. Данные возвращаются в расшифрованной форме напрямую из SSL библиотеки.

Чтобы сделать Indy TCP клиента с использованием SSL просто добавьте TIdSSLIOHandlerSocket на вашу форму с закладки Indy Misc. затем установите свойство IOHandler вашего TCP клиента в TIdSSLIOHandlerSocket. Это все, что требуется для базовой поддержки SSL. Класс TIdSSLIOHandlerSocket имеет дополнительные свойства для указания сертификатов клиентской стороны и другие расширенные SSL параметры.

18.3. Сервер SSL

Реализация SSL сервера немного более сложная, чем реализация SSL клиента. С клиентами, все что требуется это сделать хук TIdTCPClient или его наследникам к экземпляру TIdSSLIOHandlerSocket. Это происходит потому что, поскольку сервер выполняет больше работы для поддержки SSL.

Для реализации SSL сервера используется TIdServerIOHandlerSSL. TIdTCPServer's имеет свойства для установки хука на TIdServerIOHandlerSSL. Но в отличие от TIdSSLIOHandlerSocket (*Client*), класс TIdServerIOHandlerSSL требует несколько дополнительных шагов. Более конкретно - должен быть установлены сертификаты. Данные сертификаты должны быть представлены как файлы на диске и указаны в CertFile, KeyFile и RootCertFile, в соответствующих свойствах SSLOptions.

Сертификаты обычно получают из уполномоченных источников. Вы можете иметь свой собственный сертификат и своего собственно источника, но ни один из браузеров не будет доверять вашим сертификатам и браузер будет выдавать диалог с предупреждением при соединении с вашим сервером. Если вы желаете распространять через Интернет, то вы должны получить сертификат из корневого хранилища, которому стандартный браузер будет доверять. Единственный сертификат, которому доверяют все браузеры – это сертификат от Verisign. Можно также использовать сертификат от Thawte, но не все браузеры доверяют ему по умолчанию. Примечание от переводчика: самое смешное, что Thawte принадлежит Verisign

Если ваши клиенты под вашим контролем, такие как Интранет или Экстранет, то вы можете пользоваться своим собственным сертификатом. Для подавления выдачи диалога с предупреждением, вы должны установить ваш сертификат на каждый браузер, который подключается к вашему серверу. Это позволит браузеру считать ваш сертификат доверенным.

Примечание: это относится только к HTTP серверам, но SSL не ограничен использованием только в HTTP. Вы можете реализовать SSL и получить полный контроль над правилами получения и принятия сертификатов.

18.4. Преобразование сертификатов в формат PEM

Существует шанс, что вы получите ваши сертификаты формате отличном от .pem. Если это так, то вы должны преобразовать их с помощью Indy.

Данная процедура предполагает, что вы уже получили ключ и сертификатную пару от поставщика (*Certificate Authority*), например от Verisign или Thawte и уже установили их в персональное хранилище сертификатов (*Personal Certificates Store*) Microsoft Internet Explorer.

18.4.1. Экспортирование сертификата

Выберите сертификат и экспортируйте его в .pfx файл (*Personal Exchange Format*). Дополнительно вы можете его защитить с помощью пароля.

18.4.2. Преобразование файла .pfx в .pem

Как часть загрузки дистрибутива SSL, в него включена программа openssl.exe. данная программ преобразует ваш .pfx файл.

Для использование openssl.exe, используйте следующий синтаксис:

```
openssl.exe pkcs12 –in <your file>.pfx –out <your file>.pem
```

Openssl.exe запросит пароль. Введите его если он был задан или оставьте его пустым в противном случае. Также будет запрошен новый пароль для .pem файла. Это не обязательное условие, но если вы его защитите паролем, то вы должны будете создать обработчик события OnGetPassword в SSL перехватчике.

18.4.3. Разделение файла .pem

Если вы посмотрите созданный .pem файл с помощью блокнота, то вы увидите, что он разделен на две части. Эти две части содержат приватный и публичный ключи. Также приведено некоторое количество информации. Indy требует, что бы данная информация была помещена в отдельные файлы.

18.4.4. Файл Key.pem

С помощью блокнота создайте файл key.pem и скопируйте все между двумя, ниже указанными строками:

```
-----BEGIN RSA PRIVATE KEY-----
-----END RSA PRIVATE KEY-----
```

18.4.5. Файл Cert.pem

С помощью блокнота создайте файл cert.pem и скопируйте все между двумя, ниже указанными строками:

```
-----BEGIN CERTIFICATE-----
-----END CERTIFICATE-----
```

18.4.6. Файл Root.pem

Последний файл, который требуется для Indy – это Certificate Authority certificate файл. Вы можете получить его из Internet Explorer в диалоге Trusted Root Certificate Authority. Выберите поставщика (*Authority*), чей сертификат вы желаете экспортировать и экспортируйте его в Base64 (cer) формате. Данный формат, аналогичен PEM и после экспорта просто переименуйте его в root.pem

19. Indy 10 обзор

Indy 10 пока находится в стадии разработки. В ближайшие несколько недель он должен устояться. Поэтому любая приведенная здесь информация является предметом для изменения до выпуска Indy 10. Информация приведеная здесь, основывается на текущем коде, целях и направлении разработки.

Indy 10 содержит много новых возможностей, особенно относящихся к ядру. Ядро Indy 10 сделано еще более абстрактным. Ядро Indy 10 содержит много новых возможностей и улучшений в производительности.

19.1. Изменения в Indy 10

19.1.1. Разделение пакетов

Indy 10 разделена на два пакета: ядро и протоколы.

Пакет ядра содержит все части ядра, компоненты базовых клиентов и серверов. Ядро не реализует протоколы верхнего уровня.

Пакет протоколов использует ядро и реализует протоколы верхнего уровня, такие как POP3, SMTP и HTTP.

Это позволило команде Indy Pit Crew лучше сфокусироваться на специфических частях. Это также может быть полезно для пользователей, которые реализуют пользовательские протоколы и не нуждаюься в пакете протоколов.

19.1.2. Ядро SSL

Возможности SSL в Indy 10 теперь полностью подключаемые. Перед Indy 10, поддержка SSL была подключаемой на уровне TCP, в то время, как протоколы, такие как HTTP, которые используют SSL для HTTPS были вынуждены использовать Indy реализацию по умолчанию из OpenSSL.

Indy 10 продолжает поддерживать OpenSSL, тем не менее, возможности SSL в Indy теперь полностью подключаемые к ядру и уровню протоколов, что позволяет другие реализации.

В работе находятся SSL и другие методы шифрования от SecureBlackbox и StreamSec.

19.1.3. Протоколы SSL

Indy 10 теперь поддерживает неявный TLS и явный TLS в следующих клиентах и серверах:

- POP3
- SMTP
- FTP
- NNTP

Поддержка SASL кода была переработана, так что может быть использована с POP3 и IMAP4. Indy 10 теперь поддерживает анонимный SASL, плоский SASL, OTP (one-time-only password system) SASL, внешний SASL и Auth Login.

19.1.4. Клиент FTP

Клиент FTP был расширен следующим образом:

- Теперь поддержаны команды MLST и MLSD. Поддерживается стандартный формат FTP для списока директорий, который может быть легко разобран на части.
- Добавлена специальная комбинированная команда для многоблочной передачи. Примечание: это требует, чтобы сервер поддерживал команду COMB, такой как GlobalScape Secure FTP Server или серверный компонент из Indy 10.
- Добавлена команда XCRC для получения CRC файла. Замечание о поддержке см. выше.
- Клиент теперь поддерживает команду MDTM для получение даты последней модифиции
- Калькулятор OTP (One-Time-Only password) теперь встроен и OTP детектируется автоматически.
- Теперь поддержана команда FTPX или передача с сайта на сайте (когда файл передается между двумя серверами). Примечание: команда FTPX применима, только если сервер поддерживает ее (многие администраторы и разработчики теперь запрещают эту возможность по причинам безопасности).
- Добавлены специфические для FTP расширения IP6.

19.1.5. Сервер FTP

FTP теперь поддерживает:

- Команды MFMT и MFF. (http://www.trevezel.com/downloads/draft-somers-ftp-mfxx-00.html)
- Команды XCRC и COMB для поддержки режима многоблочной передачи файлов Cute FTP Pro.
- Поддержаны команды для MD5 и MMD5 (http://ietfreport.isoc.org/ids/draft-twine-ftpmd5-00.txt)
- Поддержаны некоторые ключи Unix, которые относятся к перечислению директорий, это включает ключ (-R) для получения рекурсивного списка.
- Поддержан формат Easily Parsed List файлов на сервере. (http://cr.yp.to/ftp/list/eplf.html).
- Добавлен OTP калькулятор, который может использоваться на FTP сервере.
- Компонент виртуальной системы может теперь быть использован для более легкого построения FTP сервера.
- Добавлены специфические для FTP расширения IP6.

Добавлена возможность запрещения команды FTPX. Это сделано для предотвращения нарушений защиты с использованием команд Port и PASV, причины описаны здесь:

- http://www.cert.org/tech_tips/ftp_port_attacks.html
- http://www.kb.cert.org/vuls/id/2558
- http://www.cert.org/advisories/CA-1997-27.html
- http://www.geocities.com/SiliconValley/1947/Ftpbounc.htm
- http://cr.yp.to/ftp/security.html

19.1.6. Разбор списка файлов FTP

Indy 10 содержит плагин для разбора списка файлов, который интерпретаторы почти для любых FTP серверов и даже возможно уже не функционирующих.

Если вдруг встретится не поддерживаемая система, то можно подключить пользовательский обработчик.

Поддержанные FTP сервера:

- Bull GCOS 7 или Bull DPS 7000
- Bull GCOS 8 или Bull DPS 9000/TA200
- Cisco IOS
- Distinct FTP Server
- EPLF (Easily Parsed List Format)
- HellSoft FTP Server for Novell Netware 3 and 4
- HP 3000 or MPE/iX, включая HP 3000 с Posix
- IBM AS/400, OS/400
- IBM MVS, OS/390, z/OS
- IBM OS/2
- IBM VM, z/VM
- IBM VSE
- KA9Q or NOS
- Microware OS-9
- Music (Multi-User System for Interactive Computing)
- NCSA FTP Server for MS-DOS (CUTCP)
- Novell Netware
- Novell Netware Print Services for UNIX
- TOPS20
- UniTree
- VMS or VMS (including Multinet, MadGoat, UCX)
- Wind River VxWorks
- WinQVT/Net 3.98.15
- Xecom MicroRTOS

Примечание от переводчика: они льстят себе, на самом деле список форматов примерно раз в восемь шире.

19.1.7. Прочие

Имеются и другие изменения и улучшения в Indy 10, но не ограничены:

- Были добавлены серверные перехватчики, позволяющие вам делать логи на сервере и они работают подобно перехватчикам клиента.
- Добавлены сервера и клиенты Systat UDP и TCP.
- Добавлен компонент DNS сервера.
- Добавлена поддержка HTTP соединения через прокси.
- Добавлен класс TIdIPAddrMon для мониторинга всех IP адресов и адаптеров.
- Добавлена поддержка IP6.
- Реализована система One-Time-Only password system, как в виде OTP калькулятора для клиентов, так и компонента для кправления пользователями. Имеется поддержка хэшей MD4, MD5 и SHA1.

19.2. Перестройка ядра

Ядро Indy 10 претерпело серьезные структурные изменения. Это конечно приведет к неработоспособно некоторого пользовательского кода, но изменения были сделаны так, чтобы быть как можно более совместимыми с протоколами и уровенем приложений. Иногда кажется, что команда Indy Pit Crew не заботится о конечных пользователях при внесении изменений . Тем не менее, это не так. Каждое изменение интерфейса оценивалось и взвешивалтсь доводы за и против. Изменения, которые сделаны, были разработаны так, чтобы позволить простое преобразование существующего исходного кода с минимальными усилиями. Команда Indy Pit Crew использует Indy в частных и коммерческих разработках, так что каждое изменение прочувствовала на себе.

Команда Indy Team верит, что прогресс не возможен без жертвоприношений. Благодаря небольшим изменениям в интерфейсах, было достигнуто значительное улучшение Indy. Без этих изменений, мусор бы только накапливался и будущие улучшения были бы проблематичны. Переход от Winshoes к Indy 8 и затем, Indy 8 к Indy 9 увеличил уровень абстракции. Это относится и к Indy 10.

Одним из первичных принципов Indy является убеждение, что проще программировать с помощью блокирующих режимов. Это позволяет легче разрабатывать и пользовательский код не страдает от сериализации. Цель разработки Indy это простота использования и быстрота, достаточная для 95% конечных пользователей.

Тем не менее, в больших приложениях это приводит к сильному переключению контекста и накоплению накладных расходов. Данные ограничения появляются только в больших проектах при работе свыше 2 000 конкурентных потоков, в хорошо спроектированных приложениях. В большинстве приложений ограничения в пользовательском коде возникнут раньше, чем в Indy.

Обычно для обслуживания оставшихся 5% пользователей просто написанный код должен быть заменен на сложный и тяжелый для сопровождения код, такой как ввод-вывод внахлест (overlapped IO), порты завершения ввода-вывода (*I/O completion ports*) или сильно раздробленный код, для работы с неблокирующими сокетами в потоках. Indy 10 сохраняет простоту использования модели блокирующих сокетов, достигая производительности внутри. Indy 10 делает это используя расширенные сетевые интерфейсы и эффективно транслируя это в дружественную пользователю модель блокирующих сокетов. Этим Indy 10 обеспечивает

обслуживание порядка 99.9% потребностей программистов, кроме некоторых самых необычных ситуаций.

Indy 10 достигает этого разными путями, но волокна (*fibers*) являются ключевым камнем данного достижения. Волокна очень похожи на потоки, но более гибки и имеет меньшую нагрузку, чем потоки, если использовать их должным образом.

19.2.1. Переработка обработчиков ввода/вывода (*IOHandler*)

Для получения улучшения производительности в Indy 10 были переработаны обработчики ввода/вывода и им была придана более весомая роль. Ранее роль обработчиков ввода/вывода состояла только в очень базовой обработке ввода/вывод и относилась к следующим функциям:

- Open (Connect)
- Close (Disconnect)
- Чтение сырых данных
- Запись сырых данных
- Проверка состояния соединения

Данная роль позволяла альтернативным обработчикам ввода/вывода создавать свой собственный ввод/вывод из других источников, а не только из сокетов. Даже сокет по умолчанию был реализован с помощью обработчика ввода/вывода по умолчанию.

Поскольку функциональность была минимальной, реализация обработчика ввода/вывода была очень простой. Это также часто приводило к тому что обработчики ввода/вывода вынуждены были работать не самыми эффективными методами. Например, обработчик ввода/вывода мог принимать данные из локального файла, но это не имело знания, поскольку был только один метод записи для всех данных. Даже если обработчик ввода/вывода имел возможность по быстрому чтению из файла, он не мог использовать это.

Обработчики ввода/вывода в Indy 10 реализуют не только низкоуровневые методы, но и также высокоуровневые методы. Такие высокоуровневые методы были ранее реализованы в классе TIdTCPConnection.

Как и ранее обработчики ввода/вывода могут быть созданы только посредством реализации низкоуровневых методов. Базовый обработчик ввода/вывода содержит реализации по умолчанию для методов высокого уровня, которые используют низкоуровневые методы. Тем не менее, каждый обработчик ввода/вывода может перекрыть дополнительно высокоуровневые методы для предоставления оптимизированной реализации специфичной для данного обработчика ввода/вывода.

19.2.2. Сетевые интерфейсы

Indy 9 имела только один сетевой интерфейс. В Windows этот интерфейс был Winsock и на Linux стек. Indy 10 еще продолжает поддерживать этот интерфейс, но имеет также и более эффективные интерфейсы в Windows. В данное время никаких других интерфейсов в Linux не реализовано, но возможно они появятся в будущем. Это не так важно в Linux, поскольку у него своя сетевая семантика.

Некоторые из дополнительных интерфейсов не доступны во всех версиях Windows и должны использоваться только в серверных реализациях или в системах с высоким количеством клиентских соединений. Клиенты не нуждаются в этих продвинутых возможностях.

Дополнительные интерфейсы следующие:

- Перекрытый ввод/вывод (*Overlapped I/O*)
- Порты завершения ввода/вывода (*I/O Completion Ports*)

Обычно использование перекрытого ввода/вывода и особенно портов завершения, очень сложно и требует написания заумного кода. Indy 10, как всегда позаботится обо всех деталях и предоставит разработчику дружественный интерфейс.

19.2.3. Волокна (*Fibers*)

В дополнение к расширению поддержке потоков, Indy 10 содержит и поддержку волокна. Что такое волокно? Если коротко, то это "поток", который контролируется кодом, а не операционной системой. В действительности, поток может быть трактоваться, как продвинутое волокно. Волокна подобны Unix пользовательским потокам. (От корректора: Вот и до слово блудились... Поток или волокно? Что там на у них? Сейчас уже вроде есть потоки и процессы...)

Поток – это базовая единица, которой операционная система выделяет время. Поток содержит свой собственный стек, определенные регистры процессора и контекст потока. Потоки автоматически выполняются по очередно автоматически обслуживаются операционной системой.

В целом волокна не имеют преимуществ перед хорошо спроектированным многопоточным приложением. Но волокна совмещенные с интеллектуальным планировщиком, осведомленном об особенностях их реализации, могут значительно увеличить производительность.

Множество волокон могут быть запущены в рамках одного потока. Одиночное волокно может быть запущено во многих потоках, но только в одном из них в каждый момент времени. Вы может запускать множество волокон внутри. Код выполняющий это достаточно сложный, но Indy все сделает за вас. Все компоненты Indy – клиенты и серверы поддерживают волокна, наиболее прозрачным для вас путем.

При использовании волокон, Indy так преобразовывает сложность низкоуровневых интерфейсов в дружественные пользователю интерфейсы.

Пока волокна могут быть применены только в Windows.

19.2.4. Планировщики (*Schedulers*)

Indy планирует волокна для выполнения в одином или более потоков. Волокна помещают данные в рабочую очередь и ожидают. Когда волокно заканчивает обрабатывать свои данные, планировщик помещает волокно в список доступных.

Планировщики операционной системы планирует потоки хитроумным образом, но они обладают ограниченной информацией о потоках, так как каждый поток одинаков среди всех задач системы. Планировщик операционной системы может планировать только на основе состояния ожидания и приоритета потока.

Планировщик волокон Indy (*fiber scheduler*) использует информацию специфичную для задачи, для определения нужд планировщика, приоритетов и состояний ожидания. Благодаря этому Indy в состоянии сильно уменьшить количество контекстных переключений для выполнения одной и той же работы. Это способствует увеличению производительности.

Контекст переключения – это когда один поток останавливается, а другой запускается. Для выполнения этого Операционная Система должна прерывать выполнение потока и сохранять

контекст потока, путем сохранения некоторых регистров процессора в памяти. Затем должна восстановить другой поток, загрузив ранее сохраненные регистры и передать ему управление.

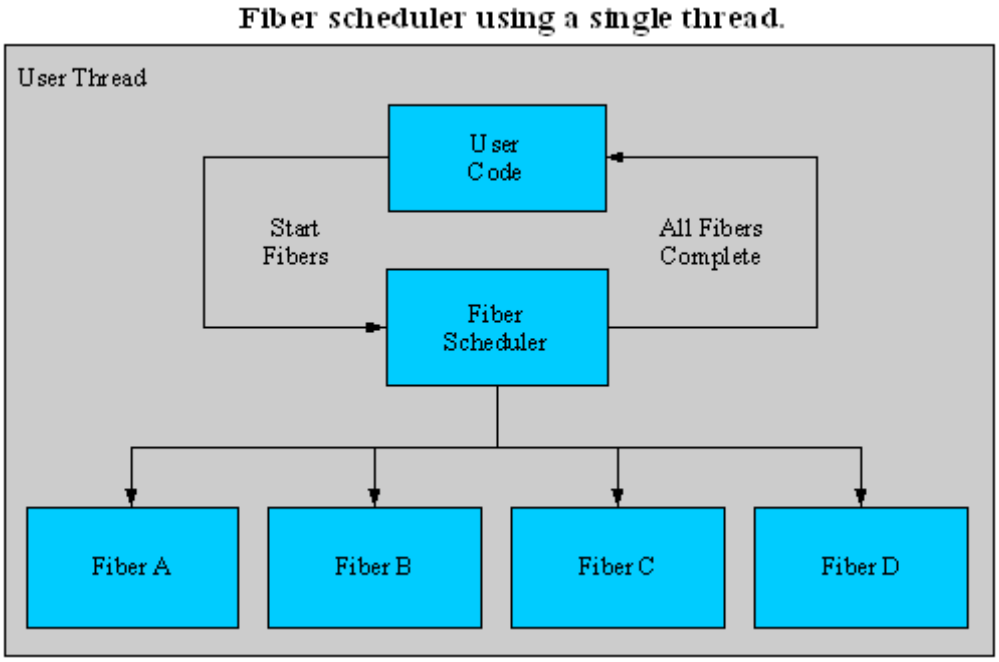
Планировщики потоков должны балансировать необходимость переключения и необходимость выделить каждому потоку достаточно процессорного времени. Частое переключение увеличивает накладные расходы на выполнение, что иногда даже превышает достигнутое увеличение производительности. Редкое переключение ведет в длятельному ожиданию потоками, замедленной реакции, потому что, потоки не получают достаточно процессорного времени.

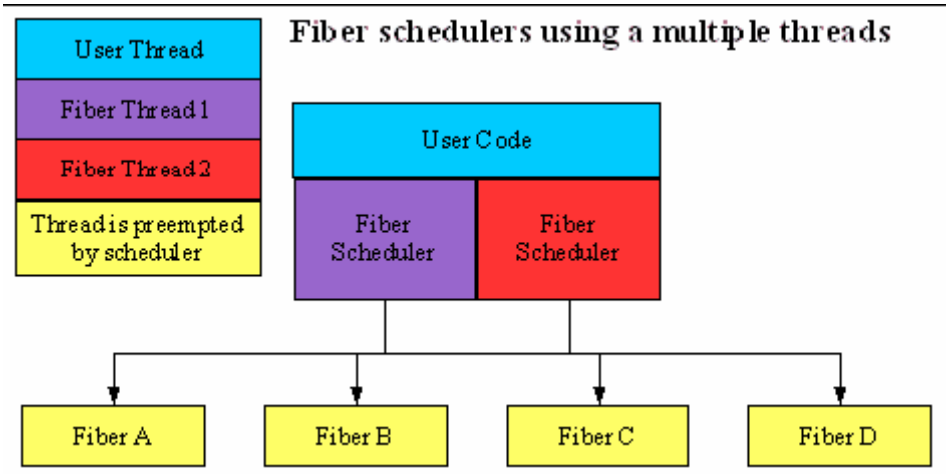
Для управления этим, Операционная Система определяет квант или максимальное время, которое требуется процессору на переключение. В большинстве случаев поток отдает управление раньше, чем наступит это время, потому что входит в состояние ожидания. Состояния ожидания случается явно или в основном неявно путем вызова операционной системы или вызова операции ввода/вывода, которая не может быть незамедлительно закончена. Когда случается такой вызов, Операционная Система воспринимает это состояние и переключается на другой поток. Если поток ожидает завершения ввода/вывода или некоторых других блокирующих операций, то он переводится в состояние ожидания и не подлежит планированию, пока запрошенная операция не будет закончена, или пока не наступит таймаут.

Планировщик Indy работает подобным образом, но определяет состояния ожидания на более высоком уровне, на основе более обширной информации. Состояние волокна может быть определено наперед, без реального переключения контекста в него и перехода в состояние ожидания. Indy также разделяет работу между волокнами и драйверами цепочек (chain engines), которые обеспечивают низкоуровневую работу.

Разделение работы позволяет использовать более эффективные сетевые интерфейсы, такие как порты завершения. Порты завершения ввода/вывода более эффективны, поскольку они работают на уровне близком к аппаратному интерфейсу. Вызовы Winsock и другие вызовы, которые далеки от аппаратного интерфейса должны общаться с ядром для выполнения действительных вызовов к аппаратному интерфейсу. Вызовы, которые общаются с ядром, должны делать переключения контекста туда и обратно. Так-что каждый вызов Winsock часто приводит к излишнему переключению контекста, только для выполнения своих функций.

Планировщики могут использовать единственный поток, множество потоков, потоки по требованию и даже пул потоков.





19.2.5. Рабочие очереди

Рабочие очереди организованы по принципу первый на входе - первый на выходе (*FIFO*), и хранят операции (*work items*), запрошенные волокнами. (Work Items скорее всего абстракция для операции ввода-вывода, так как именно они блокируются) Большинство данной функциональности полностью прозрачно для среднего разработчика, так как скрыто внутри Indy.

19.2.6. Цепочки (судя по всему внутренняя абстракция)

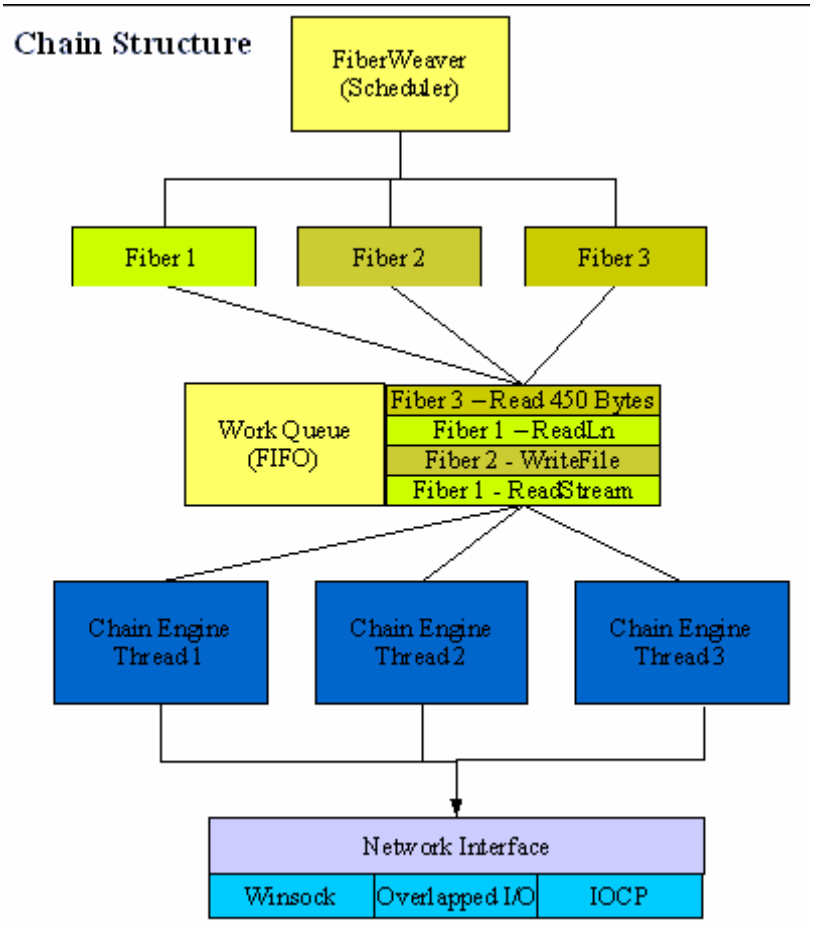
Системные рабочие очереди, планировщики и драйверы цепочек в Indy относятся к цепочкам. Несмотря на то, что цепочки используются в Indy, они не ограничены внутренним использованием в Indy и могут также использоваться в пользовательском приложении.

При использовании цепочки, IOHandler основаный на ней, помещает операцию в очередь. (От корректора: Какого хрена здесь IOHandler? Мы вроде о волокнах говорим.)

Затем, волокно приостанавливаются пока операция не обработана. Волокно может ничего не делать, пока не будет завершена операция. Каждый метод обработчика ввода/вывода выполняет одну или несколько операций. Для получения наилучшей производительности, каждый метод должен быть специализирован, настолько, насколько это возможно.

Планировщик обслуживает волокна и ожидает окончания их работы.

Драйвер цепочек запрашивает рабочую очередь и взаимодействует с планировщиком.



19.2.7. Драйверы цепочек (*chain engines*)

Драйверы цепочек – это самый низкий уровень системы цепочек. Драйвер цепочек выполняет весь действительный ввод и вывод. Драйвер цепочек может содержать один или несколько потоков.

Работа драйвера цепочек состоит в извлечении операции из рабочей очереди и ее завершения. До завершения каждой операции, драйвер цепочки оповещает планировщик и планировщик определяет, какое волокно будет обрабатываться следующим. Драйвер цепочки затем перемещается к следующей операции в рабочей очереди.

Если больше нет значений в рабочей очереди, то драйвер цепочки переходит в пассивное состояние.

Множество драйверов цепочек может быть реализовано для порта завершения ввода/вывода, Winsock, перекрытого ввода/вывода и других.

19.2.8. Контексты (*Contexts*)

В серверах Indy 9, данные специфичные для соединения были частью класса потока. Они были доступны, или через свойство thread.data, или наследованием от класса потока с добавлением новых полей или свойств для хранения. Это работало, потому что каждое соединение имело только один поток, который был специально назначен для данного соединения.

В Indy 10 сервера реализованы иначе, чем в Indy 9. потоки не ассоциируются с единственным соединением. В действительности даже не обязательно используются потоки, это могут быть волокна. Поэтому, предыдущие методы хранения данных в потоках более не приемлемы.

Indy 10 использует новую концепцию, названную контекстами. Контексты хранят специфические данные соединения и они обслуживаются и отслеживаются Indy.

20. Дополнительные материалы

Данная глава содержит дополнительные материалы, которые напрямую не относится к Indy.

20.1. Преобразование Delphi приложений в Delphi .Net

Данная глава вводит в особенности переноса существующего кода в DCCIL / Delphi. Net. Она также показывает элементы, которые больше не применимы и как их обойти, чтобы они могли правильно работать в .Net framework.

Данная статья так также дает руководство по оценке преобразования существующего приложения для работы в .Net framework с помощью DCCIL / Delphi.NET.

20.1.1. Термины

Данная статья сфокусирована на преобразовании и его последствиях для вашего Delphi кода. Поэтому, данная статья не является введением в .Net саму по себе, а только дает определение основных базовых терминов, относящихся к .NET.

20.1.1.1. CIL

Common Intermediate Language или просто Intermediate Language (*промежуточный язык*). Это язык компилятора, который преобразовывает исходный код, пригодный для использования в рантайм.

IL выполняет роль, подобную P-коду или интерпретируемому языку Java. Но также он отличается от них и реализован совсем иначе.

20.1.1.2. CLR

CLR – это сокращение от Common Language Runtime. CLR – это основа NET framework и среда исполнения приложений, которые компилируются в IL. Вначале вам может показаться, что CLR очень похожа на интерпретатор P-кода, поскольку выполняет аналогичную роль, но это не интерпретатор простого P-кода, а много больше.

20.1.1.3. CTS

CTS CLR – это сокращение от Common Type System. CTS включает предопределенные, базовые .NET типы, которые доступны в любом .NET языке. Это означает, что integer больше не определяется каждым компилятором, а встроен в CTS и поэтому integer полностью одинаков во всех .NET языках.

CTS не ограничен только integer, а включает много других типов. CTS разделяет типы на две базовые категории: значения и ссылочные типы.

Типы значений - это типы, которые записываются в стеке. Вы должны быть знакомы с термином – простой или порядковый типы. Типы значений включают integers, bytes и другие примитивы, структуры и перечисления.

Ссылочные типы – это типы, значения которых сохраняются в куче, и ссылки используются для доступа к ним. Ссылочные типы включают objects, interfaces и pointers.

20.1.1.4. CLS

CLS – это сокращение от Common Language Specification. CLS это просто спецификация, которая определяет, какие свойства языка могут и должны быть поддержаны для работы в .NET.

20.1.1.5. Управляемый код (*Managed Code*)

Управляемый код – это код который компилируется в IL и исполняется с помощью CLR. Основная цель любого .NET приложения – это быть 100% обслуживаемым кодом. (От корректора: Ну, блин сказал... ☺)

20.1.1.6. Неуправляемый код (*Unmanaged Code*)

Неуправляемый код – это откомпилированный в машинные инструкции, который получен с помощью Delphi for Windows. Неуправляемый код также включает код DLL, COM серверов и даже Win32 API. Основная цель любого .NET приложения – это не имееть такого кода.

20.1.1.7. Сборка (*Assembly*)

Сборка это коллекция .NET IL модулей. Это очень похоже на пакеты в Delphi и Delphi .NET трактует .NET сборки аналогично пакетам Delphi.

20.1.2. Компиляторы и среды (*IDE*)

Имеется несколько компиляторов и сред относящихся к Delphi и к .NET.

20.1.2.1. Компилятор DCCIL (*Diesel*)

DCCIL – это компилятор командной строки, который производит .NET вывод. DCCIL – это то, что было названо как "Delphi .NET preview compiler" и было включено в Delphi 7.

DCC – это имя стандартного компилятора командной строки в Delphi и это сокращение от *Delphi Command line Compiler*.

IL это ссылка на .NET *Intermediate Language*.

Отсюда DCC + IL = DCCIL. Произносить каждый раз D-C-C-I-L слишком громоздко и поэтому было применено имя "Diesel".

DCCIL имеет подобные DCC параметры и некоторые специфические для .NET расширения .

20.1.2.1.1 Beta

DCCIL в данный момент находится в состоянии беты и не рекомендуется для производства коммерческого кода. Поскольку в нем есть ошибки и даже незаконченные части. Но нельзя считать его бесполезным и Борланд постоянно выпускает новые версии.

Поскольку это бета, то она накладывает ограничения на распространение программ, написанных с помощью DCCIL. Любой код, который создан с помощью DCCIL, также должен распространяться как бета.

20.1.2.2. Версия Delphi 8

Borland довольно молчалив по поводу Delphi 8. Тем не менее, опираясь на публичные высказывания мы может сделать кое какие вывод насчет Delphi 8. Видно, это будет не Delphi .NET, а расширение для платформы Windows. (От переводчика: это не подтвердилось.)

20.1.2.3. Проект SideWinder

Так же есть новости о проекте SideWinder от Борланд. Конечно это не конечное имя, а кодовое имя, которое Борланд назначил для разработки.

SideWinder так же не Delphi .NET. SideWinder – это среда разработки C# для .NET, которая, будет конкурировать с Microsoft Visual Studio .NET.

20.1.2.4. Проект Galileo

Galileo – это кодовое имя Борланд для повторно используемой среды. Это первый продукт, который использует SideWinder и Galileo предназначен как базис для построения среды Delphi .NET, когда она будет выпущена.

20.1.3. Разные среды (*Frameworks*)

20.1.3.1. Среда .Net Framework

Среда .NET framework – это библиотека классов, которая является ядром .NET. Данная библиотека классов включает классы для ввода/вывода, хранения данных, простые типы, комплексные типы, доступ к базам, пользовательский интерфейс и многое другое. То чем VCL является для Delphi программистов, тем же является для .NET программистов.

АПИ Win32 ушло и было заменено классами в среде .NET, которое предоставляет лучшую и более абстрактную платформу с независимым интерфейсом. Так же предоставлен и прямой доступ до Win32 API, и до DLL. Тем не мене использования подобного доступа делает ваш код не обслуживаемым и нежелательным в .NET приложениях.

20.1.3.2. Среда WinForms

Среда WinForms – это сборка в .NET framework, которая включает классы для форм, кнопки, органы редактирования и другие элементы GUI для построения GUI приложений. Среда WinForms – это .Net управляемы интерфейс к Win32 API и – это то, что Visual Studio .NET использует для построения GUI приложений.

20.1.3.3. Библиотека времени исполнения RTL

Библиотека времени исполнения RTL содержит не визуальные низкий уровень классов в Delphi, такие как TList, TStrings и другие.

Библиотека времени исполнения RTL – также доступна и в Delphi .NET. Многое из того, что есть в RTL имеет своих двойников в .NET framework, но предоставив RTL Borland сделал более простым перенос существующего кода, без необходимости переписывать большие куски кода и позволяет создавать кросс платформенный код.

20.1.3.4. Библиотека CLX

Данная часть, вносит некоторый конфуз. До Delphi 7 было следующее разделение:

- VCL - Visual Component Library. VCL - библиотека визуальных компонент, что относилось к визуальным компонентам, к не визуальным компонентам и к RTL.
- CLX (Pronounced "Clicks") - Component Library for Cross Platform - CLX относилось к новой версии кросс платформенной части VCL, которая базировалась на QT и могла работать как в Linux, так и в Windows.

Теперь же, после выхода Delphi 7, Borland реорганизовал и переопределил значение данного акронима. Это может привести в большое замешательство, по этому примите во внимание. Начиная с Delphi 7 новое назначение следующее:

- CLX - CLX относится ко всем компонентам включенным в Delphi, C++ Builder и в Kylix.
- VCL - VCL относится к визуальным компонентам, которые работают напрямую с Win32 API.

- Visual CLX - Visual CLX относится к кросс платформенным визуальным компонентам, которые базируются на QT, доступны и в Delphi и в Kylix.
- VCL for .NET - VCL for .NET относится к новой VCL, которая запускается под .NET и предоставляет слой совместимости для старых приложений и плюс дополнительную функциональность.

Если вы посмотрите на новые определения, я не уверен, что они согласованы. Я думаю в будущем они приведут к будущим недоразумениям. Я думаю NLX (Nelix?), или NCL (Nickel?), или что ни будь еще более more совместимое будет лучшим выбором для VCL .Net. Как видим, Visual CLX – это подмножество от CLX, не VCL for .NET – это родной брат VCL, как и Visual CLX.

Это должно выглядеть так:

- VCL --> CLX
- CLX --> Visual CLX
- Visual Parts of VCL --> VCL

Хорошо, пусть это все будут мечты.

20.1.3.5. Среда VCL for .Net

Среда VCL for .NET относится к новому VCL, который работает под .NET и предоставляет уровень совместимости для старых приложений и добавляет дополнительную функциональность.

Среда VCL for .NET позволяет сделать более быстрый перенос существующих приложений, аналогично Win32 VCL и CLX. Это позволяет продолжать разработку кросс платформенных приложений. Это важное свойство, которое позволяет продолжать поддержку Windows приложений без .NET framework и также Linux.

20.1.3.6. Что выбрать WinForms или VCL for .Net?

Это область для сомнений у пользователей – должен я использовать WinForms или VCL for .NET для разработки GUI?

Следующая таблица сравнений позволит вам сделать правильный выбор. Должны быть установлены жесткие правила, но каждое приложение должно обслуживаться независимо.

VCL for .Net	**WinForms**
Большой размер дистрибутива, поскольку должен включать дополнительные сборки.	Малый размер дистрибутива, поскольку все сборки входят в состав .NET framework.
Только для платформы Win32.	Возможны подмножества для compact .NET framework for pocket PC's. можно переносить на другие реализации .NET.
Высокая степень совместимости со старым кодом.	Требуются значительные изменения в существующем коде.
Кросс платформенность с VCL for Win32 и с Visual	Только для .Net framework.

CLX for Linux (и Windows).	
Более эффективно в некоторых областях.	Не поддерживает всю оптимизацию, которая есть в VCL.
Дополнительные свойства и классы. Это включает дополнительные визуальные органы, но теряются такие вещи как списки действий (*action lists*), базы данных и многое другое.	
Не доступен полный исходный код.	Не доступен исходный код

Другая возможность – вы можете смешивать код. VCL for .NET и WinForms не исключают друг друга и могут сосуществовать в одном приложении.

20.1.4. Дополнения по переносу

Некоторые из дополнений в Delphi .NET очень важны для переноса приложений, а некоторые нет. Данные вещи не являются жизненно важными и поэтому будут рассмотрены очень кратко.

20.1.4.1. Маппирование типов в CTS

Что бы работать с .NET классами все языки должны использовать CTS (Common Type System). Delphi .NET может делать это просто, в дополнение к типам Delphi. Это может иметь место в ситуации, когда обычный Delphi код использует один набор типов, а интерфейсы к .NET использует другой набор. В результате потребуется постоянное копирование данных туда и обратно, поэтому это не очень хорошая идея с .NET. Подобная ситуация аналогична ситуацией с COM.

Ел избежание подобной проблемы, родные типы в Delphi .NET имеют их маппированые типы в CTS. Так что при объявлении Integer, это в реальности .NET Integer из CTS. Данная связь не ограничена только простыми типами, но также расширена и на объекты.

Здесь приведен список некоторых подстановок:

Delphi .Net	**Common Type System**
String	System.String
Variant	System.ValueType
Records	System.ValueType
Exception	System.Exception
TObject	System.Object
TComponent	System.ComponentModel.Component

20.1.4.2. Пространство имен (Namespaces)

Во избежание конфликтов и также как часть CLS (Common Language Specification), Delphi теперь поддерживает пространство имен. Каждый модуль теперь существует внутри пространства имен.

Когда вы видите объявление подобное следующему:

```
uses
  Borland.Delphi.SysUtils;
var
  GStatus: System.Label;
```

Важно заметить, что VCL for .NET находится в пространстве имен и влияет на директиву uses.

20.1.4.3. Вложенные типы (*Nested Types*)

Вложенные типы позволяют объявление типов внутри другого объявления типа.

20.1.4.4. Пользовательские атрибуты (*Custom Attributes*)

.Net не имеет реализации подобной RTTI в Delphi. Вместо этого она поддерживает нечто подобное, названое отражение (*reflection*). Отражение выполняет роль подобную RTTI, но функционирует немного различно. Reflection зависит от атрибутов, исполняя некоторые из его функций. Для поддержки этого Delphi .NET имеет расширение в виде атрибутов.

20.1.4.5. Другие дополнения к языку

Delphi.NET also supports many new smaller additions to the language such as static class data, record inheritance, class properties, and more. Most of the enhancements relate to features of classes at the language level.

While useful and required by the CLS, they are not essential in porting applications.

20.1.5. Ограничения

Разработка в Delphi .NET требует использования некоторых ограничений. Эти ограничения требуют, что бы код Delphi, подчинялся требованиям и ограничениям .NET.

20.1.5.1. Не безопасные элементы

В Delphi 7 появилось новое свойство, названое "Unsafe". При компилировании вашего кода в Delphi 7, вы получите предупреждение об не безопасных элементах. Не безопасные элементы – это такие элементы, которые непозволительно использовать в .NET рантайм.

Данные предупреждения включены по умолчанию и серьезно замедляют работу компилятора. Поэтому если вы не компилируете код для .NET, то вы можете их отключить. Они производят, то что я назвал эффект "C++ effect". Они замедляют компиляцию и генерируют большое количество предупреждений, что приводит к высокому соотношению сигнал-шум ".

Delphi 7 может быть использован для разработки кода, который вы желаете перенести в .NET, но DCCIL не генерирует предупреждений об небезопасных элементах. Поэтому первый шаг – это откомпилировать код в Delphi 7 и затем удалить предупреждения об небезопасных элементах.

Delphi разделяет предупреждения на три группы – небезопасные типы, небезопасный код и небезопасное приведение.

20.1.5.1.1 Небезопасные типы

Небезопасные типы включают следующее:

- Символьные указатели: PChar, PWideChar, and PAnsiChar
- Не типизированные указатели

- Не типизированные var и out параметры
- File of <type>
- Real48
- Записи с вариантными частями (Не путайте с variants)

20.1.5.1.2 Небезопасный код

Небезопасный код включает следующее:

- Абсолютные переменные (*Absolute*)
- Функции Addr(), Ptr(), Hi(), Lo(), Swap()
- Процедуры BlockRead(), and BlockWrite()
- Процедура Fail()
- Процедуры GetMem(), FreeMem(), ReallocMem()
- Ассемблерный код
- Операторы работы с указателями

20.1.5.1.3 Небезопасные приведения (Casts)

Небезопасное приведение включает следующее:

- Приведение к экземпляру класса, если он не является наследником.
- Любые приведения записей, тип record

20.1.5.2. Откидываемая функциональность (*Deprecated Functionality*)

Некоторые элементы были отброшены, так как они не совместимы с .NET и поэтому бесполезны. Многие из этих элементов вы уже знаете из ранних глав.

- Тип Real48. используйте BCD или другие математические функции.
- Функции GetMem(), FreeMem() и ReallocMem(). Используйте динамические массивы или net управление классами.
- Процедуры BlockRead(), BlockWrite(). Используйте классы из .NET framework.
- Директива Absolute
- Функции Addr и @. Используйте классы вместо блоков памяти.
- Старые тип объектов Паскаль, ключевое слово object. Используйте только ключевое слово class.
- TVarData и прямой доступ до потрохов variant. Семантика Variant поддержана, но только без прямого доступа до внутренностей.
- File of <type> - размер типов варьируется от платформы к платформе и не может быть определен во время компилирования и поэтому не может быть использован.
- Не типизированные var и out параметры. Используйте директиву const для параметра или класс родителя.

- Указатель PChar. В действительности Delphi .NET поддерживает PChar как не обслуживаемый код.
- Директивы automated и dispid. Данные директивы неприменимы в .NET.
- Директива asm – ассемблер не поддержан в .NET, код не компилируется в машинный код.
- TInterfacedObject, который включает AddRef, QueryInterface и Release.
- Динамические агрегаты – используйте implements. примечание: Implements не реализовано в текущей версии DCCIL.
- ExitProc

20.1.6. Изменения

Borland инвестировал множество ресурсов в сохранение совместимости как только это возможно. И Delphi.NET – это пока еще Delphi, но некоторые вещи не могли быть сохранены для обратной совместимости.

20.1.6.1. Разрушение (*Destruction*)

Разрушение в Delphi .NET немного отличается. Большинство кода не потребует подстройки, но важно понять в чем же различие.

20.1.6.1.1. Явное разрушение. (Deterministic Destruction)

В обычном приложении Delphi разрушение объектов делается явно. Объект разрушается только тогда, когда код явно вызовет free или destroy. Разрушение может произойти как часть разрушения собственника, но в конце все равно будет код по явному вызову free или destroy. Данное поведение называется как Решительное, явное разрушение.

Явное разрушение позволяет больший контроль, но склонно к утечкам памяти. Оно также позволяет делать ошибки, когда на разрушенный объект есть несколько ссылок или ссылка делается на другую ссылку, а о разрушении неизвестно.

Разрушение требует позаботиться об очистке объекта (finalization) явным кодом в деструкторе и освобождением памяти используемой объектом. Поэтому деструктор должен позаботиться об обеих функциях,

Программисты Delphi часто трактуют эти обе роли как одну.

20.1.6.1.2. Не явное разрушение

.Net разделяет эти функции – финализации и освобождения памяти, поскольку памятью заведует .NET. .Net использует неявное разрушение. Если вы работали с интерфейсами, то семантика подсчета ссылок, используемая в интерфейсах аналогична.

Вместо явного разрушения, когда объект сам разрушается, CLR подсчитывает ссылки на объект. Когда объект больше не используется, то он помечается для разрушения.

20.1.6.2. Сборка мусора (*Garbage Collection*)

.Net использует сборку мусора, что бы очистить память используемую объектом. Это название процесса, который определят, что объект больше используется и освобождает занятую им память.

Сборка мусора .NET очень сложная и даже базовая информация заслуживает отдельной главы, если даже не статьи.

Подобно явному и неявному разрушению, сборка мусора имеет малое влияние на перенос приложения. Поскольку для процедуры переноса, сборка мусора является ящиком фокусника, который заботится о разрушении объекта за вас.

20.1.7. Шаги по переносу

Перенос приложений в Delphi .NET для большинства приложений будет очень значимым и потребует определенной осторожности. Для большинства объектно-ориентированного кода, проще чем кажется. Данная статья не может уменьшить количество работы по переносу, но поможет вам сделать это легче. Она позволит уменьшить количество ошибок и предназначена для быстрой реализации переноса.

20.1.7.1. Удаление предупреждений (*Unsafe Warnings*)

To remove unsafe warnings, load the target project into Delphi 7. With the unsafe warnings turned on, perform a build all. Delphi will produce a series of unsafe warnings. Each warning needs to be eliminated. This may easily be the biggest step in porting your application.

20.1.7.2. Модули и пространство имен

Директивы Uses должны быть преобразованы к использованию пространства имен.

Если код требует одновременного использования в Windows и в .NET framework данные модули должны использовать IFDEF как указано. Константа CLR определена в Delphi .NET.

```
uses
  {$IFDEF CLR}Borland.Win32.Windows{$ELSE}Windows{$ENDIF},
  {$IFDEF CLR}Borland.Delphi.SysUtils{$ELSE}SysUtils{$ENDIF},
  {$IFDEF CLR}Borland.Vcl.Forms{$ELSE}Forms{$ENDIF};
```

Пространство имен будет увидено. Три главных из них – это Borland.Win32 (Windows), Borland.Delphi (RTL) and Borland.VCL (VCL for .NET).

20.1.7.3. Преобразование DFM

Delphi for .NET пока не поддерживает формат DFM. Это возможно будет в будущем, так что данный шаг требуется если вы желаете использовать бета версию DCCIL.

Поскольку DFM не поддержаны, все формы должны быть сконструированы с помощью кода. Есть также программа разработанная для выполнения данной задачи, которая может быть использована с нормальным приложениями Delphi.

20.1.7.4. Преобразование файла проекта

Application.CreateForm более не поддержан, так что формы должны быть созданы вручную, как обычные компоненты. Для установки главной формы приложения, установите свойство Application.MainForm перед вызовом Application.Run.

20.1.7.5. Разрешение с различиями в классах

Во время компиляции могут проявиться различия между VCL for .NET и VCL, поскольку появились небольшие различия между VCL и Visual CLX. Каждое из этих различий должно быть разрешено с помощью IFDEF.

20.1.7.6. Нужна удача

Если вам повезло, то ваш проект преобразован и работает нормально.

Перенос – это не только перекомпиляция и потребует некоторого времени. Перенос в .NET требует подобных усилий, которые нужны для переноса в из VCL приложений в Visual CLX. Хотя время и усилия значительны, но это все равно меньше, чем писать с нуля и позволяет делать кросс платформенную разработку, так как код будет повторно использоваться и там и там, конечно если он спланирован должным образом.

20.1.8. Благодарности

Я использовал много источников. Я извиняюсь, если я кого-то забыл указать в благодарностях. Я желаю поблагодарить следующих товарищей: John Kaster, Brian Long, Bob Swart, Lino Tadros, Danny Thorpe, Eddie Churchill, Doychin Bondzhev.

21. Об авторах

21.1. Chad Z. Hower a.k.a Kudzu

Чад Хувер (Chad Z. Hower), известный так же, как "Kudzu" является автором и координатором проекта Internet Direct (Indy). Indy состоит из более, чем 110 компонент и является частью Delphi, Kylix и C++ Builder. В опыт Чада входит работа с трудоустройством, безопасность, химия, энергетика, торговля, телекоммуникации, беспроводная связь и страховая индустрии. Особая область Чада это сети TCP/IP, программирование, межпроцессорные коммуникации, распределенные вычисления, Интернет протоколы и объектно-ориентированное программирование.

Когда он не занимается программирование, то он любит велосипед, каяк, пешие прогулки, лыжи, водить и просто чем ни будь заниматься вне дома. Чад, чей лозунг "Программирование это искусство а не ремесло", также часто публикует статьи, программы, утилиты и другие вещи на Kudzu World http://www.Hower.org/Kudzu/.

Чад как истинный американский путешественник, проводит свое лето в Санкт-Петербурге, Россия, а зиму в Лимасоле на Кипре.

Чад доступен через Веб форму.

Чад является ведущим разработчиком в Atozed Software.

21.2. Hadi Hariri

Hadi Hariri – это старший разработчик и ведущий проекта в Atozed Computer Software Ltd. (http://www.atozedsoftware.com/) и также помощник координатора проекта Internet Direct (Indy), Open-source проект TCP/IP компонент, который включен в Kylix и Delphi 6. раньше работал для ISP и в программисткой компании, он имеет твердые знания Internet и клиент-серверных приложений, как сетевой администратор и администратор по безопасности. Hadi женат и проживает в Испании, где он главный авто для журнала по Delphi и участник Borland Conferences и пользовательских групп.