1. **Основы систем, использующих транзакции. Свойства транзакций.**

Транзакция - группа операций (эффект от выполнения группы операций) со свойствами:

* Атомарность
* Согласованность (до и после выполнения транзакции система будет находиться в непротиворечивом состоянии, транзакция правильная)
* Изолированность (для каждой транзакции создается видимость того, что данная транзакция в системе выполняется одна)
* Устойчивость (если транзакция завершилась успешно, гарантируется, что ни при каких обстоятельствах изменения от транзакции не могут быть потеряны)

ACID - atomicity, consistency, isolation, durability.

Управление транзакцией:

* BEGIN TRANSACTION
* COMMIT
* ROLLBACK - выполняется приложением при необходимости, системой после восстановления от сбоя

Доступность - доля времени работы системы с приемлемым временем ответа, измеряется в %.

Доступность = MTTF/(MTTF+MTTR)

MTTF - среднее время наработки на отказ

MTTR - среднее время восстановления

Mean Time To Failure(Recovery)

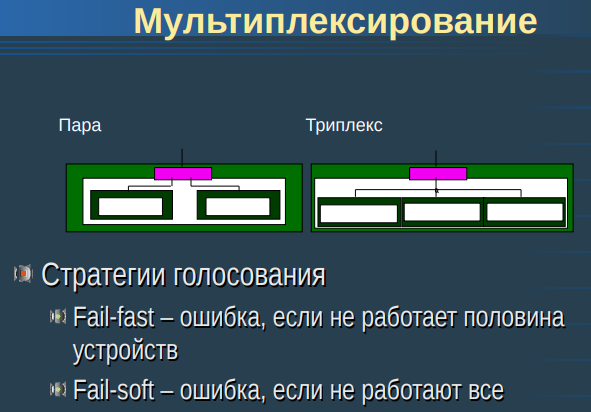
1. **Подход к отказоустойчивости для аппаратного обеспечения, схемы дублирования (N-плексирование)**

Надежность - это

* Достоверность
* Целостность
* Доступность
* Безопасность

Техники обеспечения отказоустойчивости:

* FAIL FAST MODULES: работай или остановись - короткое время обнаружения ошибки, основная задача - улучшение MTTF и MTTR, передача сообщений на уровень выше
* SPARE MODULES: короткое время восстановления, один работает второй запасной
* INDEPENDENT MODULE FAILS: путем дизайна MTTFPair ~ MTTF2 / MTTR - модули работают параллельно, результаты сравниваются
* MESSAGE BASED OS: изоляция ошибок, нет разделяемой памяти - система обрабатывает сообщения между модулями, если модуль не отвечает, отправляет сообщение повторно / перезапускает его
* SESSION-ORIENTED COMM: достоверные сообщения(обнаружение пропавших и дублирующихся сообщений) - TCP/IP
* PROCESS PAIR: маскирование проблем АО и ПО - дублирование
* TRANSACTIONS: предоставление ACID свойств



Из NOFAIL FAST MODULES можно сделать FAIL FAST MODULES.

При чтении можно выполнять операцию параллельно, однако устройство голосования всё ещё может дать ошибку.

Плата за FAIL FAST MODULES – уменьшение MTTF. (Если у каждого устройства по отдельности MTTF 1 мес, то у пары – 2 недели)

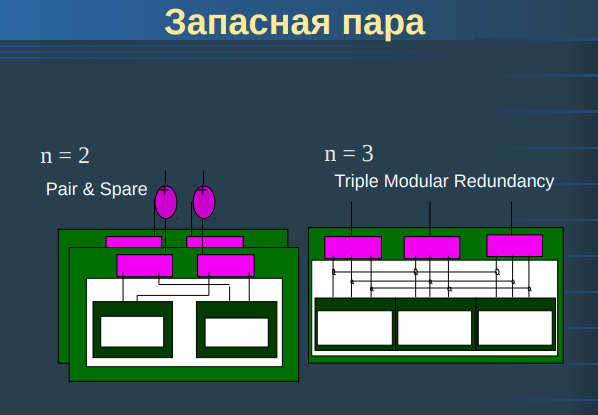
1. **Подход к отказоустойчивости для аппаратного обеспечения, схемы дублирования (Запасная пара)**

Надежность - это

* Достоверность
* Целостность
* Доступность
* Безопасность

Техники обеспечения отказоустойчивости:

* FAIL FAST MODULES: работай или остановись - короткое время обнаружения ошибки, основная задача - улучшение MTTF и MTTR
* SPARE MODULES: короткое время восстановления, один работает второй запасной
* INDEPENDENT MODULE FAILS: путем дизайна MTTFPair ~ MTTF2 / MTTR - модули работают параллельно, результаты сравниваются
* MESSAGE BASED OS: изоляция ошибок, нет разделяемой памяти - система обрабатывает сообщения между модулями, если модуль не отвечает, отправляет сообщение повторно / перезапускает его
* SESSION-ORIENTED COMM: достоверные сообщения(обнаружение пропавших и дублирующихся сообщений) - TCP/IP
* PROCESS PAIR: маскирование проблем АО и ПО - дублирование
* TRANSACTIONS: предоставление ACID свойств



На верху стоит обязательно fail-soft(при отключении первой пары, вторая продолжит работать)

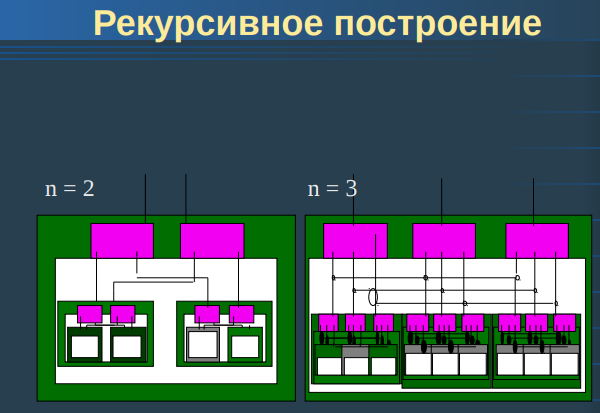
1. **Подход к отказоустойчивости для аппаратного обеспечения, схемы дублирования (Рекурсивное построение)**

Надежность - это

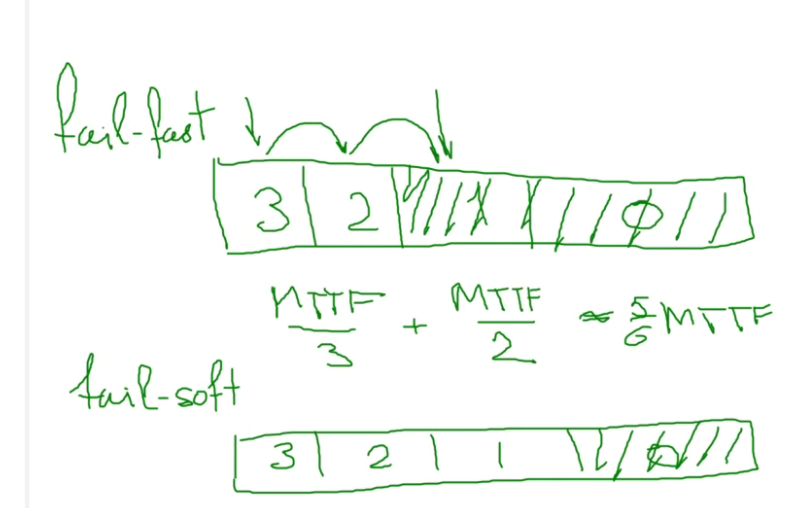
* Достоверность
* Целостность
* Доступность
* Безопасность

Техники обеспечения отказоустойчивости:

* FAIL FAST MODULES: работай или остановись - короткое время обнаружения ошибки, основная задача - улучшение MTTF и MTTR
* SPARE MODULES: короткое время восстановления, один работает второй запасной
* INDEPENDENT MODULE FAILS: путем дизайна MTTFPair ~ MTTF2 / MTTR - модули работают параллельно, результаты сравниваются
* MESSAGE BASED OS: изоляция ошибок, нет разделяемой памяти - система обрабатывает сообщения между модулями, если модуль не отвечает, отправляет сообщение повторно / перезапускает его
* SESSION-ORIENTED COMM: достоверные сообщения(обнаружение пропавших и дублирующихся сообщений) - TCP/IP
* PROCESS PAIR: маскирование проблем АО и ПО - дублирование
* TRANSACTIONS: предоставление ACID свойств



Устройство предлагает два выхода от устройств голосования

11/6 для fail-soft

1. **Методы обеспечения надежности хранения информации;**

* Операции READ/WRITE
* Параметры
  + Адрес блока (reliability)цилиндр, поверхность, дорожка, сектор)
  + Содержимое (reliability)набор байт)
* Ошибки
  + Нарушение целостности
  + Ошибка адресации
  + Физическое повреждение

Необходимая избыточность

* Для данных
  + Контрольная сумма(позволяет определить ошибки согласованности и целостности)
  + Адрес вместе с данными(предотвратить ошибки механики по возвращению данных не из того адреса)
  + Копия данных
* В логике
  + Скрытое контрольное чтение после записи
  + Скрытое восстановление во время чтения(вернуть пользователю свежие данные и скрыто обновлять диск на котором остались старые)

Запись на два диска должна осуществляться последовательно чтобы сохранить исходную копию данных.

1. **Методы обеспечения надежности выполнения процессов;**

Обязательно хорошее и корректное проектирование, кодирование и тестирование

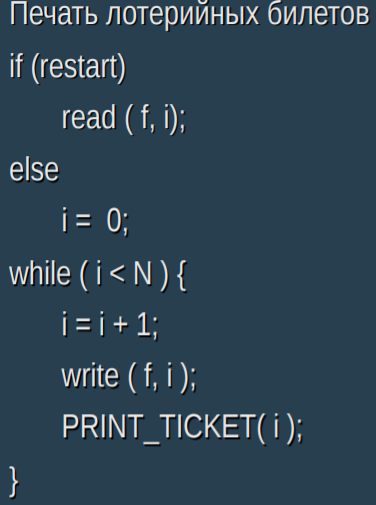
Техника обеспечения отказоустойчивости:

* Модульность (изоляция, ограничение распространения ошибок)
* Разнородный дизайн
* Программирование N-версий (разная реализация)
* Защитное программирование ( проверка параметров данных)
* Аудитор: Проверка структур данных в фоновом режиме
* Транзакции ( восстановление состояния после ошибки)

Необходимо fail-fast программное обеспечение

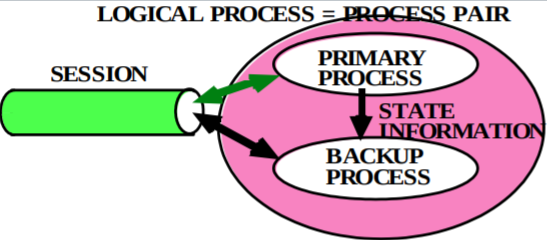
Процессы с хранением состояния

* Состояние процесса:
  + Локальные и глобальные переменные
  + Содержимое стека
  + Регистры
  + Текущая выполняемая команда
* Постоянная запись состояния во внешнюю память
* Считывания состояния при перезапуске
* Перезапуск при сбое



* Достоинства
  + Простая реализация
* Недостатки
  + Медленная работа
  + Проблема синхронизации логики приложения с последним состоянием
  + Скорость восстановления зависит от скорости ввода/вывода

Пара процессов

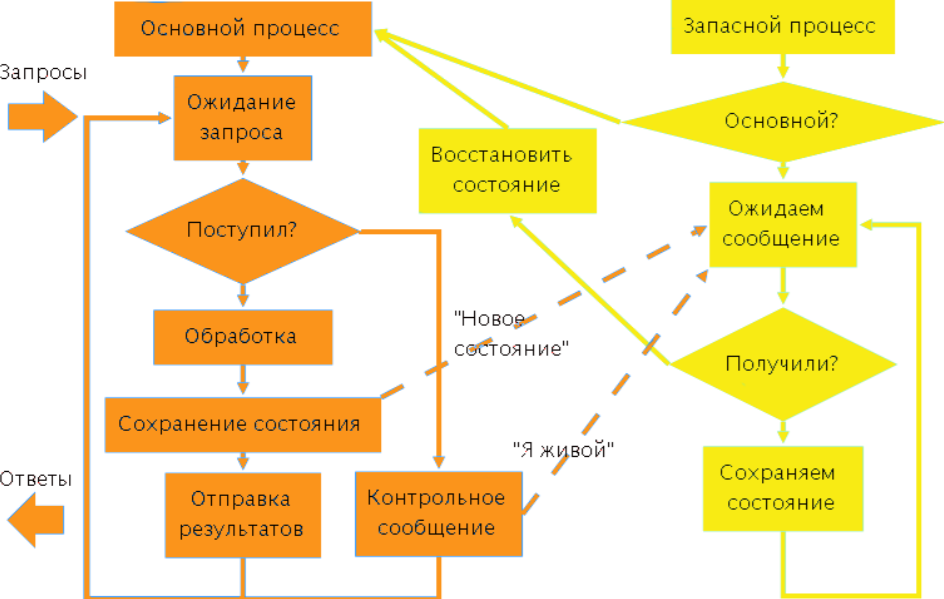


1. Процесс постоянно подает сигналы, что он работает
2. Резервному передавать состояния

Если основной процесс рушится, то запасной занимает его место

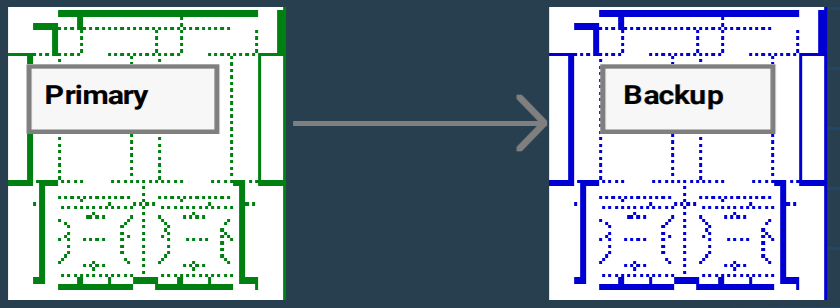
Сбои

1. После обработки
2. сохранение состояния
3. отправка результатов



* Достоинства
  + Высокая скорость переключения
  + Возможно масштабирование
* Недостатки
  + Высокая нагрузка на сеть
  + Повторная отправка ответов

Общая концепция.



* Программы, данные, процессы реплицируются в 2-х узлах
* Пара ведет себя как единая система
* Система становится логической концепцией как процесс
* Сохранение журнала транзакций на Backup узле
* Переключение на Backup в случае сбоя

1. **Методы обеспечения надежности передачи данных;**

OSI - Open Standart Institute

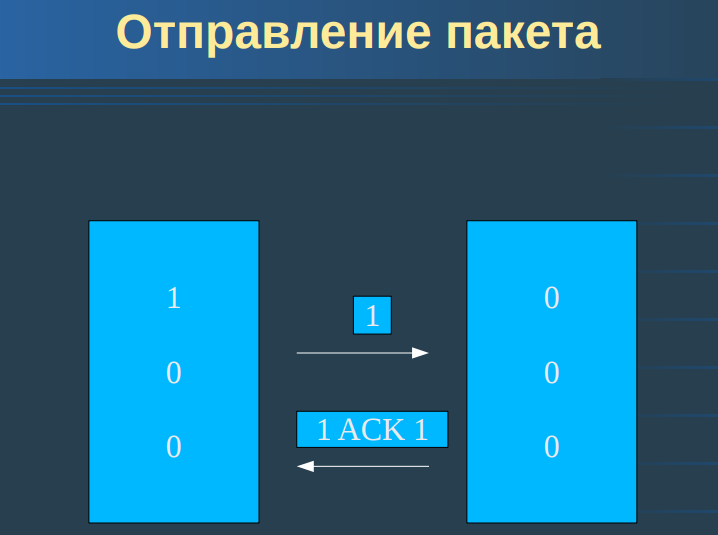


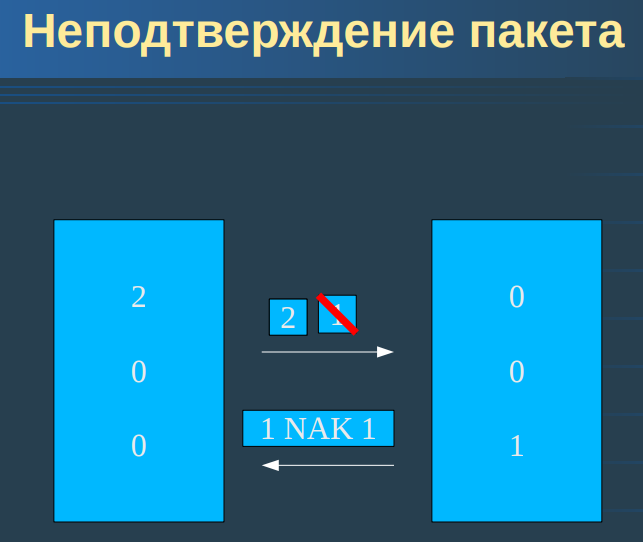
1. Какие физ. характеристики что означают (0/1)
2. Пакеты. МАС адрес у пользователя ( устройство). Как понять, что пакет начался/закончился. Может передать пакет по лок. сети(широковещательное), по МАС адресу, либо всем
3. Можно передавать информацию м/у любыми абонентами, м/у кот. есть путь. Адрес назначает админ  
   если пакет потерялся, то никто не узнает(?вот тут хз)
4. Передача данных  
   TCP ( более ресурсоемкий) или IPX
5. Логическое соединение установлено, но TCP соединение устанавливается только когда реально начинается общение
6. Передача инфы в родных форматах гарант того, что будет прочитана корректно
7. Действия необходимые для решения проблем/задач приложения

Нумерация пакетов:  
Узел А, узел В

* Номер последнего пакета
* Номер последнего подтвержденного пакета
* Номер последнего полученного пакета

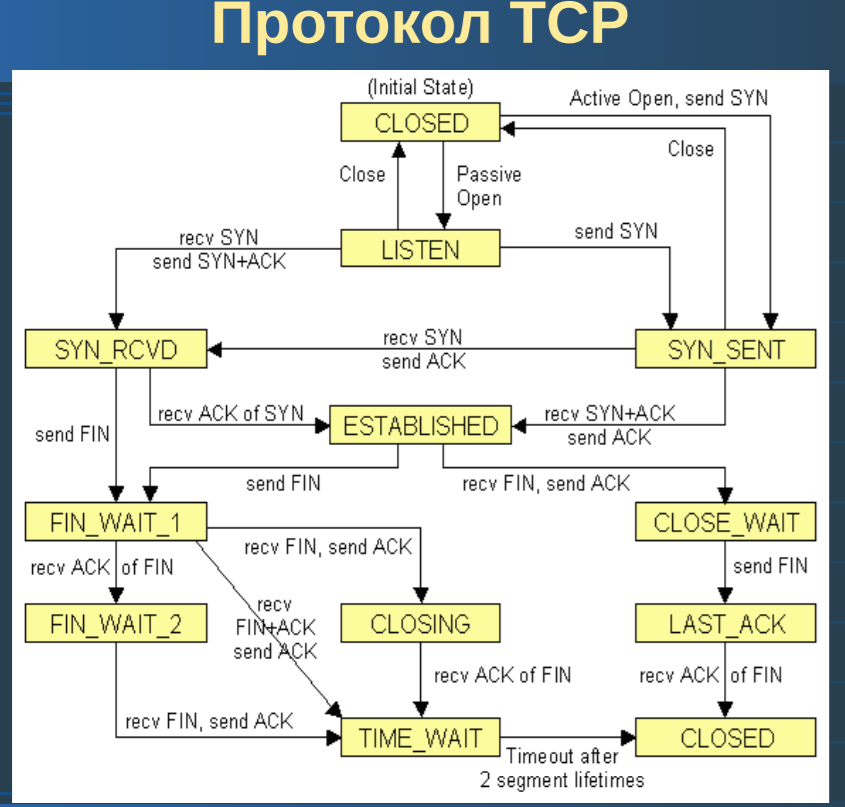
При передаче пакетов, они остаются в памяти





Возможная оптимизация:

* выборочные подтверждения
  + не получены 1,6,7 пакеты
* Групповое подтверждение
  + получены все пакеты до 5 включительно
* Динамическое изменение окна протокола ( кол-во неподтвержденных пакетов до ожидания)



3 пакета для синхронизации

1. **Атомарные действия. Классификация действий. Обеспечение атомарности.**

Нет изменений в системе (READ) => атомарность не нужна  
Запись 1 блока на диск может быть прервана.

1. Обычная запись - атомарность отсутствует
2. Запись с контрольным чтением (защита от физ. повреждений)  
   атомарности нет (отключ. свет)
3. Дублированная запись (зеркалирование)  
   -параллельно писать нельзя  
   -непараллельно (2 ситуации) - есть атомарность  
   сначала на А, потом на В - в 2 раза медленнее

Минус: двойная стоимость

1. Журнализированная запись - файл последовательного доступа  
   Инфа для восстановления записывается в журнал
   1. Ставим отметку - начало операции

Сохраняем старое значение блока

* 1. Модификация диска
  2. В журнал - операция завершена

Сбой во время записи => перезапуск системы, смотрим журнал  
Нет отметки о завершении => возвращаем старое значение

Классификация действий

1. Незащищенные действия

* Отсутствуют все свойства транзакций, кроме непротиворечивости
* Не являются атомарными
* Эффект от выполнения не является зависимым
* Являются обратимыми

*Примеры:*Операция WRITE  
Операция DRAW LINE  
Просверлили дырку - необратима  
insert -> commit - обладает свойствами транзакции

1. Защищенные действия

* Имеют ACID свойства
* Не объявляют своих результатов до завершения (скрывают изменения)
* Изменения согласовано управляются
* Могут быть автоматически откатаны в случае ошибки
* Не могут быть откатаны в одностороннем порядке после успешного завершения (нельзя сделать ROLLBACK после COMMIT; либо хотим аннулировать после объявления результатов - не ок)

*Примеры:*Операторы DML в языке SQL

1. Реальные (настоящие) действия

* Происходят в реальном мире (нельзя вдвоем просверлить 1 дырку)
* Имеют CI свойства
* Необратимы (в основном)

*Примеры:*Пуск ракеты  
Сверление дырок  
  
Правила использования  
1. Незащищенные действия включаются в состав защищенных более высокого уровня (СУБД сама справляется)  
2. При реализации атомарных действий необходимо знать о механизмах отката нижнего уровня (когда пишем блок, нужно знать, как восстанавливать)  
Не нужно: запись на SSD (не перепис. старое значение)  
3. Реальные действия откладываются до момента, когда откат невозможен.  
4. Защищенные действия – основные строительные блоки приложения

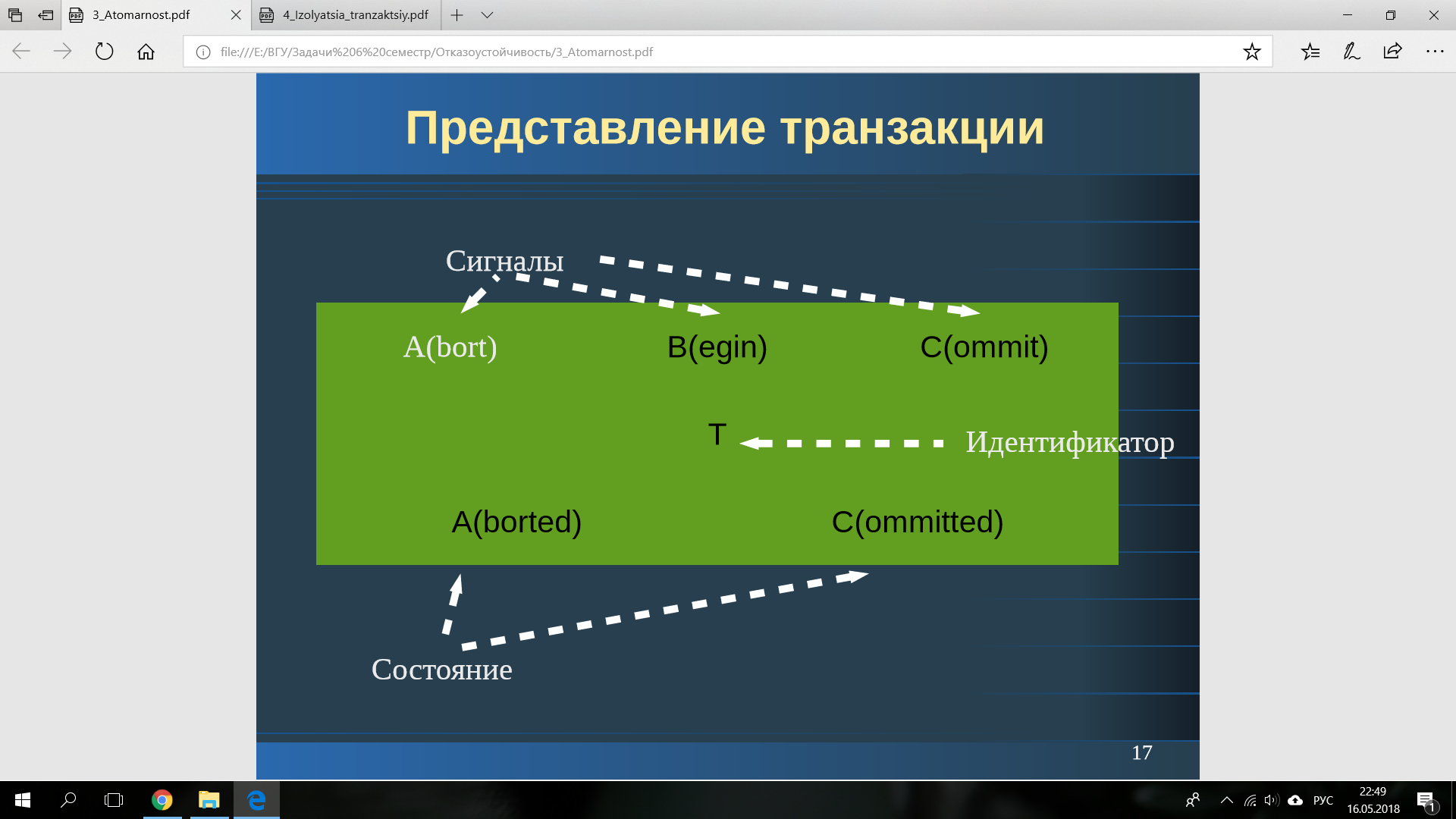
1. **Модели транзакции. Свойства транзакций. Примеры использования в различных приложениях.**

Транзакции - группа операций со свойствами: ACID (Atomicity, Consistency, Isolation, Durability). Упомянутые выше свойства означают следующее:

* *Свойство атомарности* (Atomicity) выражается в том, что транзакция должна быть выполнена в целом или не выполнена вовсе.
* *Свойство согласованности* (Consistency) гарантирует, что по мере выполнения транзакций данные переходят из одного согласованного состояния в другое — транзакция не разрушает взаимной согласованности данных.
* *Свойство изолированности* (Isolation) означает, что конкурирующие за доступ к базе данных транзакции физически обрабатываются последовательно, изолированно друг от друга, но для пользователей это выглядит так, как будто они выполняются параллельно.
* *Свойство долговечности или устойчивость* (Durability) трактуется следующим образом: если транзакция завершена успешно, то те изменения в данных, которые были ею произведены, не могут быть потеряны ни при каких обстоятельствах (даже в случае последующих ошибок).

Управление транзакциями:

1. Begin transaction - сообщает о начале транзакции.
2. Commit - фиксирует все изменения, которые производились в БД в процессе выполнения транзакции.
3. Rollback - оператор отката всей транзакции (может быть выполнен либо приложением, либо система автоматически осуществляет восстановление после сбоя, для всех транзакций, что не завершились).

****

Виды транзакций:

1. Плоские транзакции
2. Плоские транзакции с контрольными точками
3. Плоские транзакции сохраняемыми с контрольными точками
4. Цепочные транзакции
5. Вложенные транзакции
6. Многоуровневые транзакции

Примеры использования в различных приложениях:

Примеры приложений:

* Резервирование билетов
  + Купить билет МОСКВА – С.ПЕТЕРБУРГ
  + Купить билет С.ПЕТЕРБУРГ – КАЛИНИНГРАД
  + Вернуть билет(!)
  + Купить билет МОСКВА - ХЕЛЬСИНКИ
* Массовое обновление
  + 1 000 000 000 записей
  + Обновление процентов
    - UPDATE ACCOUNTS set
    - ACCOUNT=ACCOUNT\*1.055
  + Сбой при обновлении 999 999 999 записи
  + Откат всей полезной работы и потеря времени на сам откат
* Распределенные изменения
  + Перевод денег между банками
    - БД1: update ACCOUNTS set ACCOUNT=ACCOUNT-171 where ID=1657;
    - БД2: update ACCOUNTS set ACCOUNT=ACCOUNT+171 where ID=6571;
    - БД1: COMMIT, БД2: COMMIT.

1. **Плоские транзакции;**

Плоские транзакции (Flat) - это основные строительные блоки для реализации принципа атомарности; иначе говоря, выделение некоторой последовательности действий в виде плоской транзакции обеспечивает принцип "все или ничего".

По мере того как данные и вычисления становятся все более распределенными, атомарность плоских транзакций становится значительным неудобством. Согласно правилам обработки плоских транзакций, либо должны успешно завершиться все компоненты глобальной транзакции, либо не должна завершиться ни одна из них.

Минимальный набор стандартных операторов плоской транзакции:

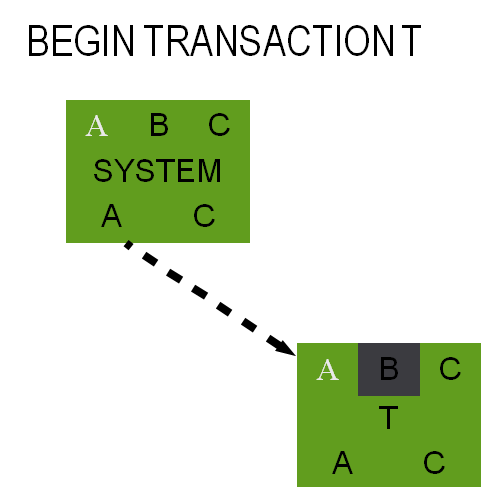
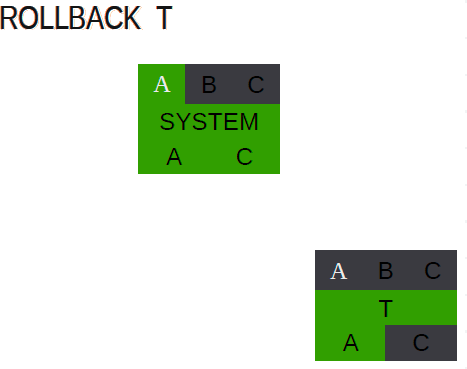
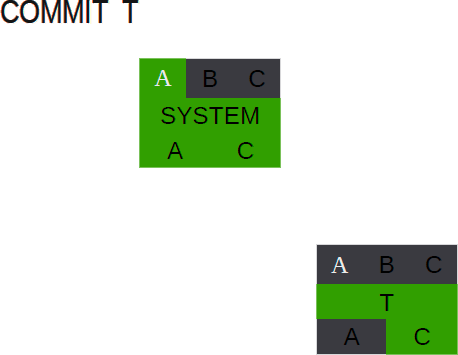
* Begin transaction
* Commit
* Rollback

Только один уровень управления: COMMIT или ROLLBACK

Нельзя выполнить:

* Частичный ROLLBACK
* COMMIT за несколько шагов

Причины появления других моделей

* Сложная логика приложений
* Распределенная и параллельная обработка данных 

1. **Плоские транзакции с точками сохранения;**

К минимальному набору стандартных операторов плоской транзакции добавляются:

* SAVEPOINT T, <SP> - фиксирует текущее состояние транзакции;
* ROLLBACK T, <SP> - откат до состояния <SP>.

В случае сбоя транзакция откатывается целиком.

Достоинства:

* возможность частичного отката;
* каждая элементарная транзакция является плоской.

Недостатки:

* потеря всей работы при сбое на операции массового обновления.

Пример:



Примеры использования: система покупки билетов, задачи с рекурсивным поиском оптимального варианта

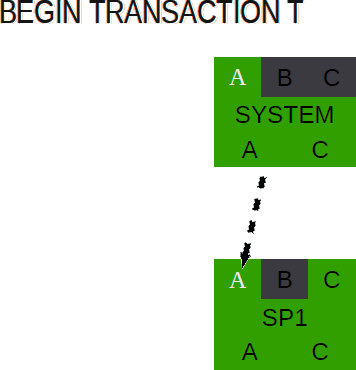
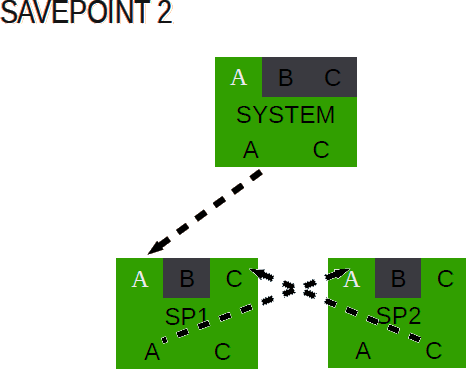
1. **Сохраняемые точки сохранения;**

* При выполнении SAVEPOINT информация о контрольной точке сохраняется во внешнюю память.
* Откатывается только последний незавершенный интервал работы.
* При перезагрузке после сбоя устанавливается состояние последней точки сохранения.
* Phoenix transaction.

Достоинства:

* Возможность продолжить работу с последней точки сохранения.

Недостатки:

* Запись во внешнюю память.
* Необходимо API для получения текущего состояния
* Необходима дополнительная логика синхронизации приложения и БД.

1. **Цепочные транзакции;**

CHAIN ≈ COMMIT + BEGIN TRANSACTION

(предыдущий фрагмент транзакции откату не подлежит).

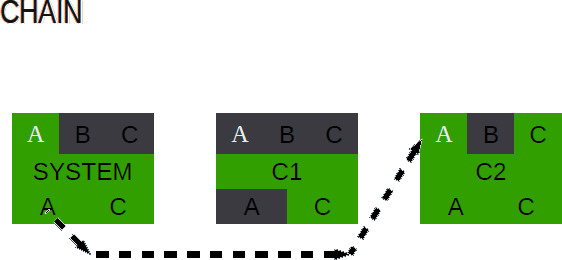
При старте новой транзакции гарантируется, что другая транзакция не сможет занять ранее заблокированные данные/объекты. Транзакция передаёт зарезервированные ресурсы.

Каждый фрагмент - плоская транзакция.

Достоинства:

* Есть возможность продолжить работу.

Недостатки:

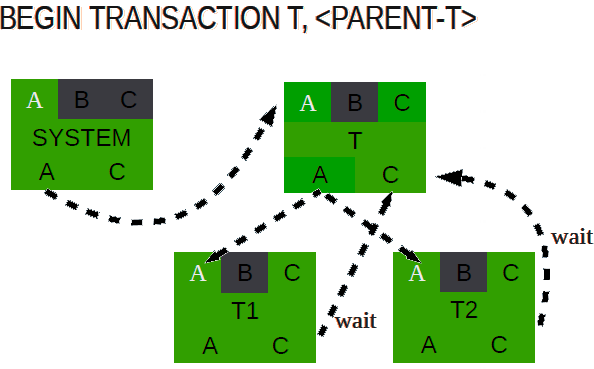
* Нужна синхронизация приложения.
* Откатить все нельзя, дополнительное восстановление за счет логики приложения.

1. **Вложенные транзакции, эмуляции с использованием точек сохранения;**

Особенности вложенных транзакций:

* Обобщение контрольных точек
* Организация транзакций в виде дерева
* Нижний уровень – плоские транзакции
* Вложенные транзакции – дерево транзакций, в котором каждое поддерево вложенная или плоская транзакция
* Листья дерева – плоские транзакции
* Транзакция в корне дерева – основная головная транзакция, остальные - субтранзакции
* При выполнении субтранзакцией COMMIT выполнение откладывается до COMMIT транзакции-предка
* Выполнение ROLLBACK вызывает откат всех субтранзакций
* Изменения, сделанные в транзакции, видны всем субтранзакциям
* Субтранзакции одной транзакции выполняются параллельно и не видят изменений друг друга
* Субтранзакции теряют свойство durability

Всех, кому был интересен этот профиль, приглашаю в свой паблик https://vk.com/amountofexpressions

****

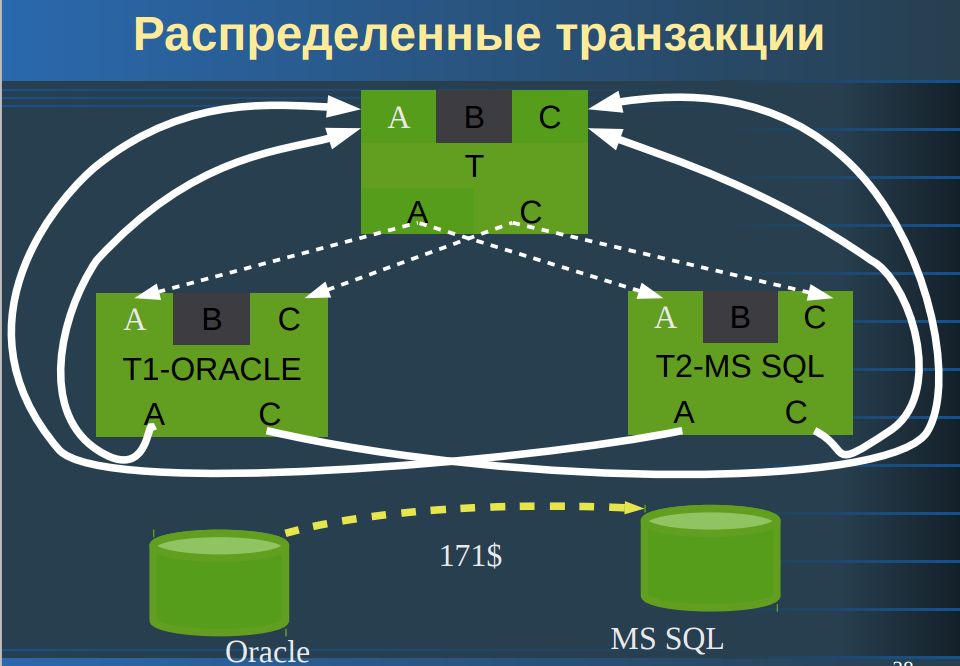
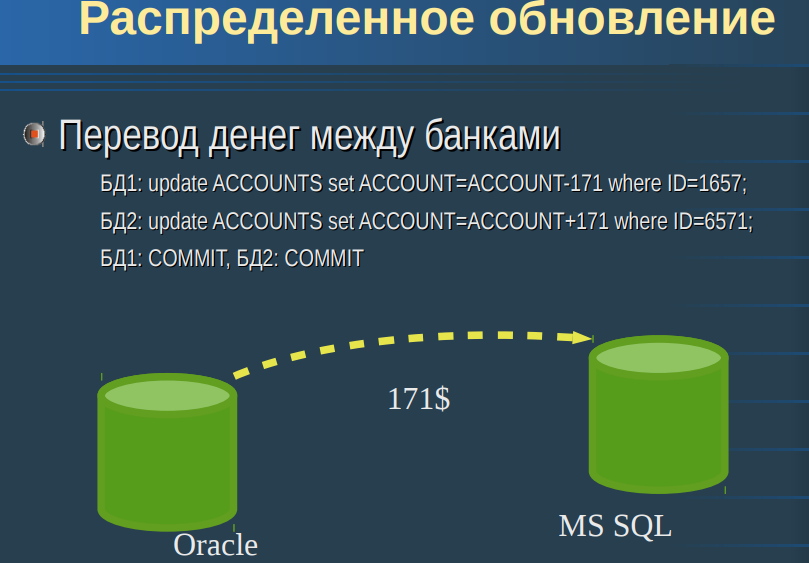
Достоинства:

* Поддержка приложений со сложной структурой
* Поддержка параллельной обработки данных с единым механизмом согласования

Если обработка выполняется последовательно, то можно эмулировать с помощью контрольных точек.

Реализуется с использованием компенсационных транзакций на каждом уровне вложенности

1. **Распределенные транзакции;**

****

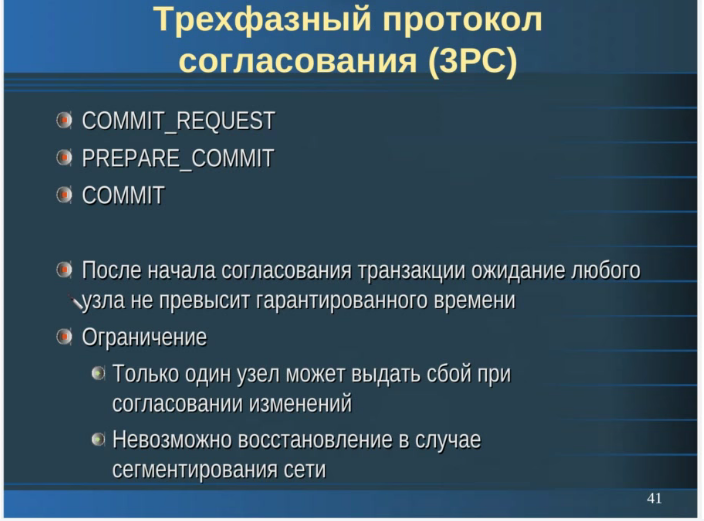
Распределенной называется транзакция, охватывающая операции нескольких взаимодействующих компонент распределенной системы. Каждая из этих компонент может работать с какими-либо СУБД или иными службами, например, использовать очереди сообщений, или даже работать с файлами. При откате транзакции все эти операции должны быть отменены. Для этого необходимо выполнение двух условий:

промежуточная среда должна поддерживать управление распределенными между несколькими компонентами транзакциями;

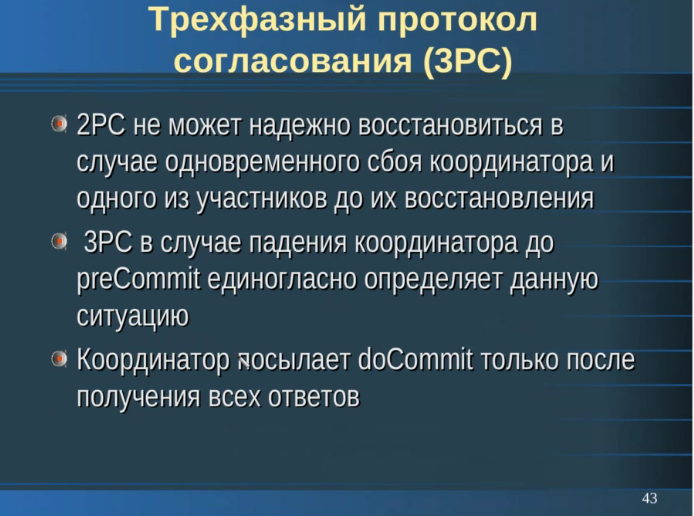
компоненты распределенной системы не должны работать с какими-либо службами или ресурсами, которые не могут участвовать в транзакции.

****

Координатор хранит в себе транзакцию и отвечает за выполнение транзакции всеми участниками. При соглашении всех участником координатор будет пинговать каждого участника, пока тот не подтвердит выполнение.   
Минус: приложение может зависнуть в случае если один из участников отвалится ожидая его восстановления.



Штатная работа: при первой фазе мы ждём подтверждения всех участников, если кто-то не успевает согласиться, откатываем всё. Схема позволяет избежать состояния когда кто-то из участников отказал и база находится в неопределенном состоянии.

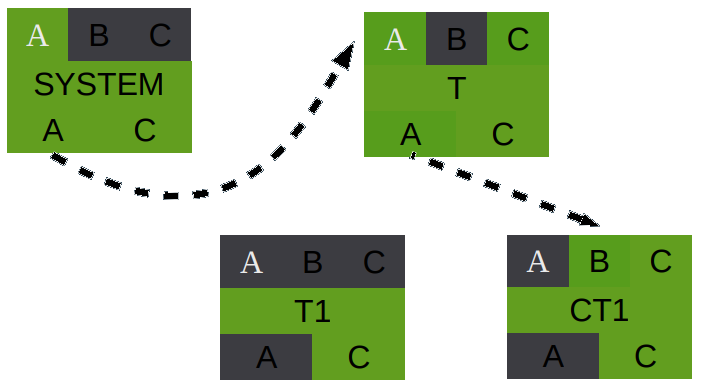


1. **Многоуровневые транзакции;**

Согласование в несколько этапов:

* PRECOMMIT - раннее согласование с возможностью отката ( Изменения не видны другим транзакциям). Для T1 создаётся компенсационная транзакция CT1 которая хранит действия, обратные Т1(insert-delete)
* COMMIT - окончательная фиксация изменений. Иначе применение CT1

PRECOMMIT T1



1. **Изоляция транзакций. Методы реализации совместного доступа.**

Изолированность – создание видимости, что в системе одновременно выполняется только одна транзакция, скрытие проблем совместного выполнения и упорядочивание транзакций в процессе выполнения.

Блокировка – механизм разграничения доступа к объектам. При обращении к объекту транзакция

Статическое планирование, динамическое планирование.

1. **Зависимости между транзакциями;**

* 2 операции: READ + WRITE
* 4 варианта совместного выполнения 2-х разных транзакций транзакций
* T1 READ A; T2 READ A – нет проблем
* Версии объекта <0,1>,<O,2>...<O,n>
* Если транзакция выполняет READ <O,i>, то она зависит от версии объекта
* Если транзакция выполняет WRITE <O,i>, то версия объекта зависит от транзакции

1. **Возможные конфликтные ситуации;**

2 параллельно работающие транзакции, еще выполняются

1. Потерянные изменения (lost update)

T2 READ <c,1>  
T1 WRITE <c,2> (действия перезаписались)  
T2 WRITE <c,3>

1. Грязное чтение (dirty read)

T2 WRITE <c,2>  
T1 READ <c,2> (прочли неокончательные данные)  
T2 WRITE <c,3>

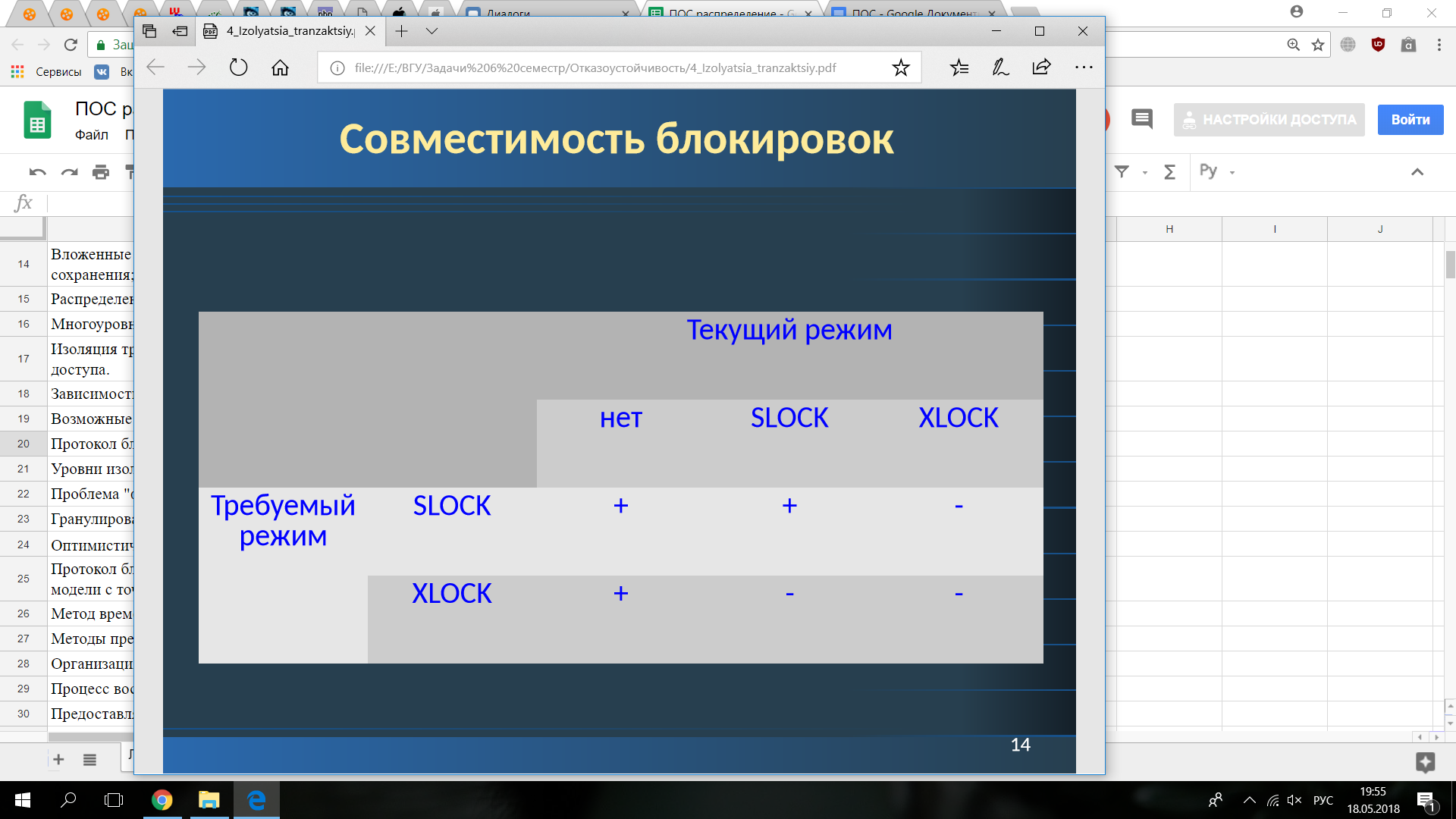
1. Неповторяемое чтение (unrepeatable read)

T1 READ <c,1>  
T2 WRITE <c,2>  
T1 READ <c,2>

Теория сериализации

1. Наличие циклов в потоках данных между транзакциями  
   приводит к конфликтным ситуациям
2. При отсутствии циклов возможно упорядоченное  
   выполнение транзакций без конфликтов
3. Если транзакция правильно сформированная и двухфазная, то все истории выполнения являются допустимыми
4. **Протокол блокировок, история выполнения транзакций;**

* 3 оператора для работы с блокировками:
  + LOCK READ (SHARED LOCK)
  + LOCK WRITE(EXCLUSIVE LOCK)
  + UNLOCK
* Правильно сформированная транзакция – если всем операциям чтения предшествует операция блокировки на чтение, аналогично для записи.
* Двухфазная транзакция – если в процессе присутствует фаза, когда блокировок только увеличивается, а потом фаза, когда только уменьшается.





Выполнение операций в 3 столбце недопустимо, т.к. блокировки будут ожидать друг друга

1. **Уровни изоляции;**

* 0: Транзакции устанавливают short xlocks (правильно сформированная относительно write, не двухфазная)
* 1: Транзакции устанавливают long xlocks (правильно сформированная относительно write, двухфазная)
* 2: Транзакции устанавливают short slocks (правильно сформированная, не двухфазная)
* 3: Транзакции устанавливают long slocks (правильно сформированная, двухфазная)

Сравнение уровней изоляции:



Уровни изоляции в SQL:

SET ISOLATION LEVEL TO

[ READ UNCOMMITED | (1)

READ COMMITED | (2)

REPEATABLE READ | (2.99)

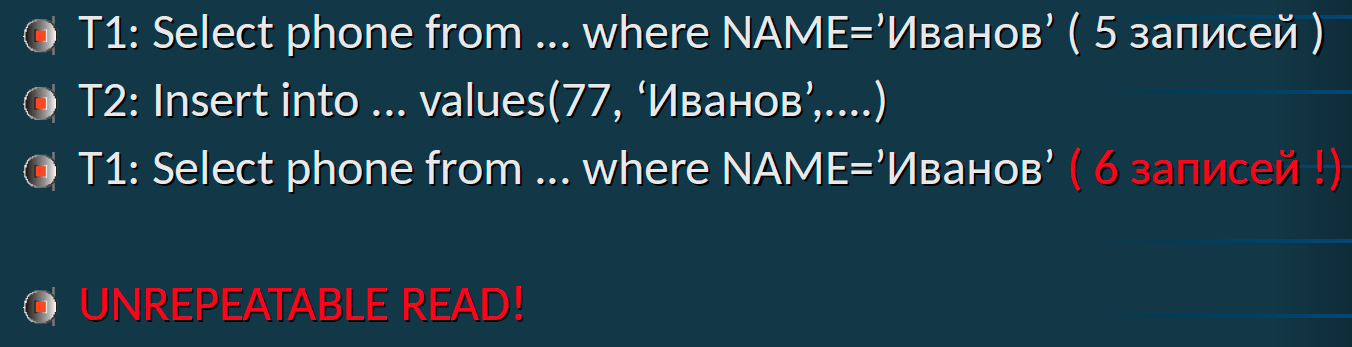
SERIALIZABLE ] (3)

OPEN CURSOR C - “cursor stability”

Длительность блокировки зависит от реализации.

1. **Проблема "фантомных" записей, предикатные блокировки;**

Нельзя поставить блокировку на несуществующие (новые, insert) данные.

(2.99) - имеет эту проблему с insert, update => unrepeatable read

(3) - не имеет этой проблемы, конфликтов не будет

Предикатные блокировки:

Предикатное условие - мн-во записей, удовлетворяющих условию.

Перед выполнением SELECT выставляется блокировка. Пришла новая транзакция. Для того, чтобы определить, можно ли выполнить действие, нужна проверка совместимости предикатов:

непересекающиеся мн-ва => транзакция допустима.

пересекающиеся => вторая транзакция должна быть приостановлена, смотрим тип блокировки.

Проверка пересечения - перебор предикатов.

“+”: Решение проблем фантомных записей

“-”:

* Сложное выделение предикатов
* Сложность задачи определения совместимости предикатов (NP-полная задача)
* Пессимистичное поведение (записей может и не быть в БД, а транзакция будет ждать)

1. **Гранулированные блокировки;**

Страницы - размер приближен к размеру блока, ввод/вывод → быстрее.

+ Одинаковый размер позволяет избежать фрагментации.

Идея: при доступе делать блокировку и сообщать об этом более верхнему уровню (там тоже блокировка?) => можно выполнять действия со страницей.

SIX - текущий объект блокируется S, а нижестоящий - X

Совместимость блокировок:

U - update блокировка

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | | Текущий режим | | |
| U | S | X |
| Требуемый режим | U | - | + | - |
| S | - | + | - |
| X | - | - | - |

Порядок блокирования:

* От корня дерева к узлам
* Если S блокировка, то вышестоящая как минимум IS ( IX, S, SIX, U, X)
* Если X блокировка, то вышестоящая как минимум IX (SIX, U, X)
* Блокировка обновления Update
* Порог эскалации блокировок = N

Эскалация блокировок – это процесс, при котором множество блокировок с маленькой гранулярностью, конвертируются в одну блокировку на более высоком уровне иерархии с большей гранулярностью.

“+”

* Решение проблемы фантомных записей
* Меньше ресурсов при массовых обновлениях

1. **Оптимистичные и пессимистичные методы блокирования;**

Пессимистичные - высокая вероятность обращения к одним и тем же данным разными транзакциями.

Оптимистичные - низкая вероятность работы с одними и теми же данными.

Отличия:

* завершаются всегда / могут не завершаться;
* блокируют все записи на все время работы транзакции / блокируют только для проверки

Hot Spot записи в БД

Оптимистичные методы - нацелены на использование коротких по времени блокировок.

Снимки значений:

* сохранить старое значение каждого объекта;
* откладываем обновление до фазы 2;
* фаза 1 COMMIT: если значение поменялось, то откат, иначе блокируем XLOCK
* фаза 2 COMMIT: выполняем все изменения, снимаем все блокировки

“+” эффективное использование ресурсов

“-” в процессе выполнения транзакции не видно изменений

Отметки времени:

* Сохранить время обновления каждого объекта
* Откладываем обновления до фазы 2
* Фаза 1 COMMIT: если время поменялось, то откат, иначе блокируем XLOCK
* Фаза 2 COMMIT: выполняем все изменения, снимаем все блокировки

Многоверсионность:

* Запись R хранит историю изменений

R: [T0..T1): V0, [T1…T2) V1, [T2….) V3.

* Ti – время выполнения COMMIT транзакцией
* Последняя версия в история – актуальная версия
* При старте транзакции запоминаем время t
* При чтении транзакция читает значения записей на момент t
* При изменении записи создается частная версия
* При выполнении COMMIT выполняется согласование

!!!(Более подробно об отметках времени и многоверсионности смотри в соответствующем билете)

1. **Протокол блокирования для вложенных транзакций и для модели с точками сохранения;**
2. **Метод временных меток, многоверсионность;**

**Метод временных меток**

Метод, хорошо работающий в условиях редких конфликтов транзакций и не требующий построения графа ожидания транзакций.

Основная идея состоит в следующем: если транзакция T1 началась раньше транзакции T2, то система обеспечивает такой режим выполнения, как если бы T1 была целиком выполнена до начала T2.

Для этого каждой транзакции T предписывается временная метка t, соответствующая времени начала T. При выполнении операции над объектом r транзакция T помечает его своей временной меткой и типом операции (чтение или изменение).

Перед выполнением операции над объектом r транзакция T1 выполняет следующие действия:

* Проверяет, не закончилась ли транзакция T, пометившая этот объект. Если T закончилась, T1 помечает объект r и выполняет свою операцию.
* Если транзакция T не завершилась, то T1 проверяет конфликтность операций. Если операции неконфликтны, при объекте r остается или проставляется временная метка с меньшим значением, и транзакция T1 выполняет свою операцию.
* Если операции T1 и T конфликтуют, то если t(T) > t(T1) (т.е. транзакция T является более "молодой", чем T1), производится откат T и T1 продолжает работу.
* Если же t(T) < t(T1) (T "старше" T1), то T1 получает новую временную метку и начинается заново.

К недостаткам метода относятся потенциально частые откаты транзакций. Кроме того, в распределенных системах не очень просто вырабатывать глобальные временные метки с отношением полного порядка (это отдельная большая наука).

Но в распределенных системах эти недостатки окупаются тем, что не нужно распознавать тупики, а построение графа ожидания в распределенных системах стоит очень дорого.

**Многоверсионность**

При любой модификации создается новая версия (перечень отличий от оригинальной версии). При старте Ti транзакции в качестве идентификатора запоминается время ее старта ti. При чтении транзакция читает значения записей на момент ti. При выполнении COMMIT происходит согласование (проверяем, можно ли сделать актуальной): ставим короткую блокировку на чтение последней актуальной версии, если она та, от которой мы порождены, то COMMIT, иначе - ROLLBACK.

1. **Методы предотвращения взаимного блокирования;**

* Соблюдать один порядок блокировки (например, по алфавиту - BLOCK A, B, C ...)
* Если большая вероятность изменений, сразу вешать ULOCK:

T1 ULOCK A

T1 READ A

T1 WRITE A

T1 UNLOCK A

* Блокировка при хождении по курсору:

SELECT …

WHERE ID = 11

FOR UPDATE;

1. **Организация журнала**

* Журнал – файл, в который постоянно идёт запись, последовательный файл с записями обо всех изменениях
* Допускает восстановление любой версии объекта
  + Мягкий сбой(выключение питания)
  + Жесткий сбой(повреждение накопителя)
* Write Ahead Log запись перед действием
* Запись в журнал без кэширования
* Предоставление единого интерфейса подсистемам
* Несколько копий журнала
* Цикличное использование или архивация из-за растущего размера журнала
* Пользователи журнала – ресурсный менеджер и менеджер транзакций и архивации.
* Запись блоками фиксированного размера
* Возможна потеря последнего блока данных

Устройство

* LSN(log sequence number) – однозначно определяет номер записи(не хранится)
* PREV\_LSN – предыдущий номер записи
* RM\_ID (Resource manager id)
* TR\_ID (Transaction\_id)
* TR\_PREV\_LSN – предыдущий номер записи для данной транзакции
* UNDO/REDO информация для отката
* Место под файлы Log не должно заканчиваться

Статья алгоритмы ARVES

1. **Процесс восстановления**

Журнал делиться контрольными точками, которые показывают какая часть журнала не нужна на этапе восстановления.

Контрольная точка

* Предоставляет ссылку на запись журнала и хранится в первом блоке журнала
* Соответствует моменту времени когда все изменения данных для операций журнала записаны на диск
* Записывается при штатном завершении работы системы
* Записывается периодически(допустим при простое)\
* Оптимизация REDO
  + Периодическая запись в файлы минимальных LSN которые соответствуют страницы кэша, содержащие минимальные изменения. (записывать номер меньше которого нам не надо смотреть записи)
* Оптимизация UNDO
  + Поставить указатель об успешном завершении UNDO чтобы перепрыгнуть на запись к которой мы откатились

1. **Предоставляемые функции**

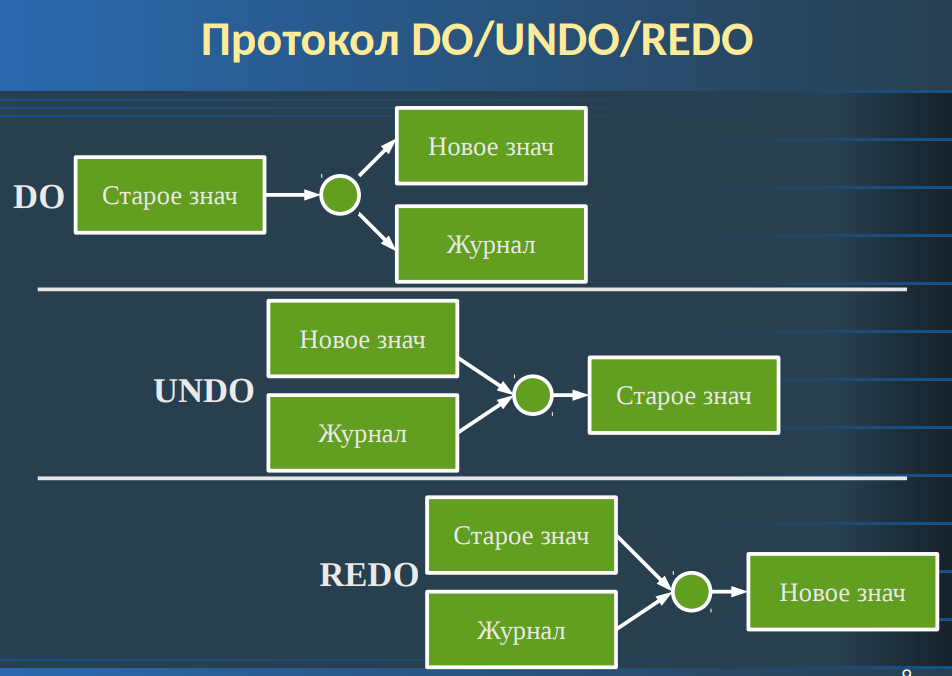
Nested Top Actions

1. **Логика рестарта**

Записывать чаще контрольные точки.

1. **DO/UNDO/REDO протокол**

* Фаза DO соответствует стандартному режиму работы(журнал растёт)
* Фаза Undo соответствует моменту, когда система получается rollback, либо он происходит автоматически после сбоя
* Redo - восстановление системы после жёсткого сбоя(например изменения из памяти не успели перенестись на диск после коммита). Позволяет докатить изменения, которые не успели примениться и перенестись на диск.

****

1. **Проблема многократного REDO/UNDO**

* Update ... set A=A+2 where ...
* Повторное выполнение приводит к проблеме

Как узнать записаны ли изменения в таблице данных на диск?

Страницы с данными помечаются номерами LSN того, кто последний раз менял данные. Каждая запись в журнале транзакций однозначно идентифицируется регистрационным номером транзакции в журнале (номером LSN). Регистрационные номера транзакций в журнале упорядочены таким образом, что если два изменения описываются записями в журнале, помеченными номерами LSN1 и LSN2, а LSN2 больше LSN1, то изменение, помеченное номером LSN2, произошло после изменения LSN1.

1. Находим контрольную точку восстановления (информация обычно в заголовке)
2. От нее идем в направлении конца файла и пытаемся сделать REDO. В памяти формируем список активных транзакций
3. Дошли до конца. В памяти останется только список незавершенных транзакций
4. Идем обратно и производим откат для оставшихся в списке

При проходе по REDO смотрим на LSN страниц данных. Если LSN страницы журнала <= LSN страниц данных, изменения не выполняются.

Если при UNDO LSN страницы журнала > LSN стр данных изменения не выполняются.

Передвигаем контрольную точку восстановления при успешном восстановлении.

1. **Физическая/логическая журнализация**
2. Физическая журнализация

Все данные рассматриваются как набор байтов на диске

В журнал записываются:

* Смещение в файле
* Кол-во байт
* Новые значения байт
* Старые значения байт
* Простая реализация
* Большой объем информации при небольшом логическом изменении (перезапись бОльшей части В-дерева)

1. Логическая журнализация

Если в REDO, то пишем в UPDATE SET A = A + 2

Если в UNDO, то пишем в SET A = A - 2

При восстановлении системы непонятно, какие подсистемы завершились, а какие нет, и что делать - REDO или UNDO

В журнал записываются:

* Имя и параметры операции
* Обратная операция
* Компактный размер журнала
* Не все операции имеют обратные
* Нарушение целостности при частичном выполнении составной операции

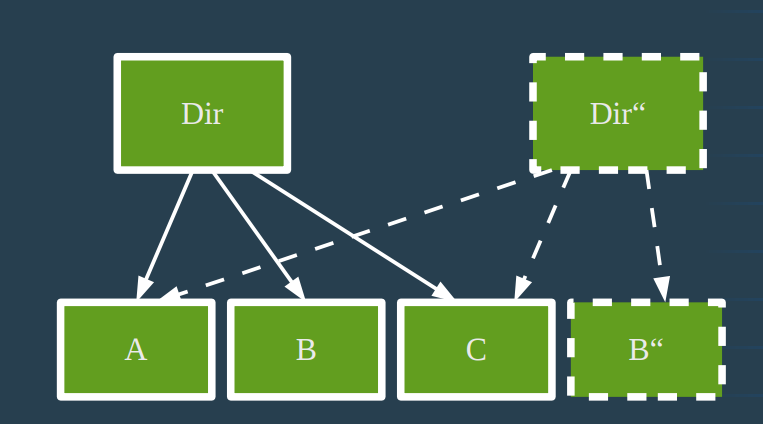
1. Смешанный вариант  
   В журнал записываются:

* Низкоуровневая операция
* Обратная низкоуровневая операция

*Примеры:*  
ADD RECORD ...  
INSERT INDEX RECORD ...

1. **Теневая журнализация**

Аналог многоверсионности. Файл из блоков А В С  
Хотим заменить В на В’(содержимое файла)



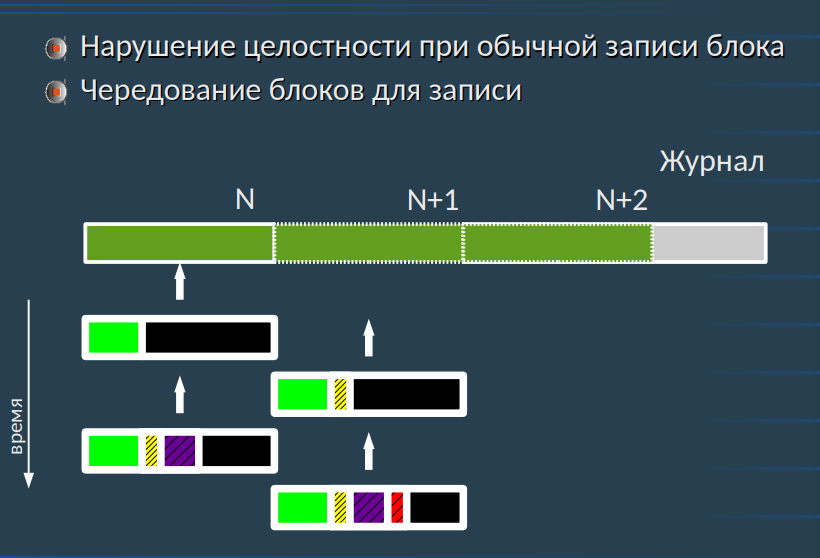
Создается копия, которая видна только мне

в журнал записываются только моменты переключения, когда создавалась новая копия и какие файлы в нее включены.

1. **Оптимизация работы с журналом. Алгоритм 'ping-pong'**

Алгоритм пинг-понга решает проблему записи последней страницы журнала, когда на диске могут не оказаться операций завершающей транзакции

Пишем в N блок, затем в N+1, потом в N

Каждая следующая запись не туда, когда была запись последняя  


1. **Оптимизация работы с журналом. Алгоритм WADS**

WADS Write Ahead Data Set

Проблема при записи на диск система должна получить 4 параметра: номер сектора, номер дорожки, номер головки, номер пластины

Алгоритм пытается убрать время позиционирования, которое мы ждем, пока диск докрутится до головки и там будет нужное место под ней(паразитное ожидание)

Запись размера блока в В выполнится на 1й диск 1ю дорожку в 1й попавшийся сектор  
Следующая запись на 1й диск 2ю дорожку в 1й попавшийся сектор  
Дойдем до последней дорожки, пойдем в обратном направлении  
Как только блок завершен, он записывается на постоянное место хранения на устройство А

при сбое и восстановлении будем искать на В последнюю(по времени) запись(просто перебором дорожек и смотрим на дату и время записи)

Минимальный размер В = 2 страницы ввода/вывода  
используя тот же принцип можно вместо В взять какое-нибудь немеханическое устройство

Неэффективное выполнение операции записи для механических устройств Компенсация ожидания за счет дополнительного устройства

1. **Оптимизация работы с журналом. Оптимизация доступа к последней странице журнала**

2 варианта алгоритма записи в последнюю страницу журнала**:**

1 Вариант:

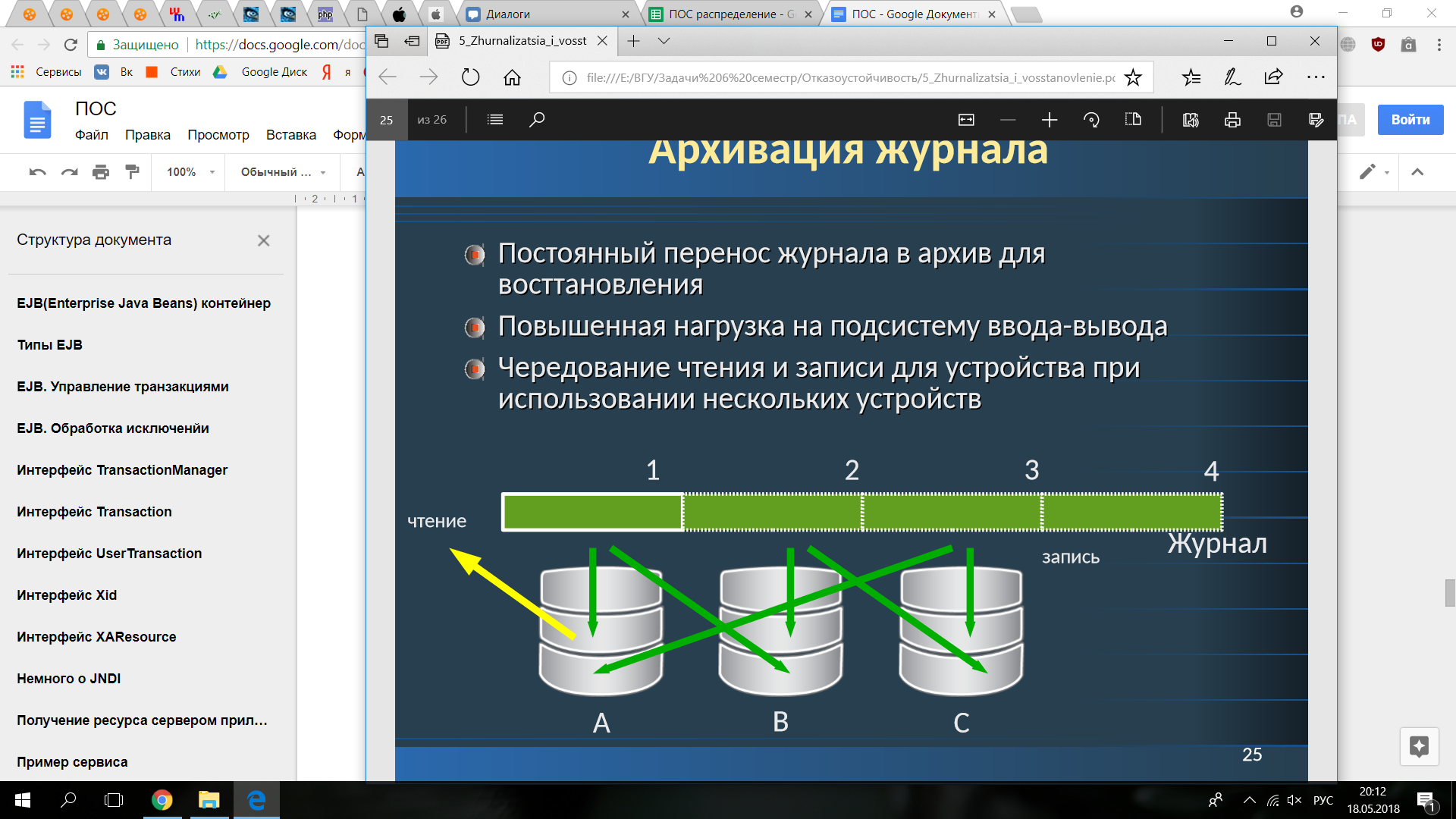
* заблокировать последнюю страницу
* дописать данные на страницу памяти
* записать страницу на диск
* снять блокировку

2 Вариант:

* заблокировать последнюю страницу
* дописать данные на страницу памяти
* записать страницу на диск
* снять блокировку

1. **Буферизация журнала**
2. **Архивация журнала**

* Постоянный перенос журнала в архив для восстановления
* Повышенная нагрузка на подсистему ввода-вывода
* Чередование чтения и записи для устройства при использовании нескольких устройств

****

1. **Оптимизация работы с журналом. Неоткатываемые атомарные действия (NTA).**
2. **Оптимизация работы с журналом. Работа с журналом при восстановлении системы. CLR записи.**
3. **Управление транзакциями в приложениях. Границы транзакций для EJB компонентов. Типы атрибутов транзакций. Обработка исключений.**

JDBC. Управление транзакциями

*try {*

*connection.setAutoCommit(false);*

*…....*

*connection.commit();*

*} catch(Exception e) {*

*connection..rollback();*

*} finally {*

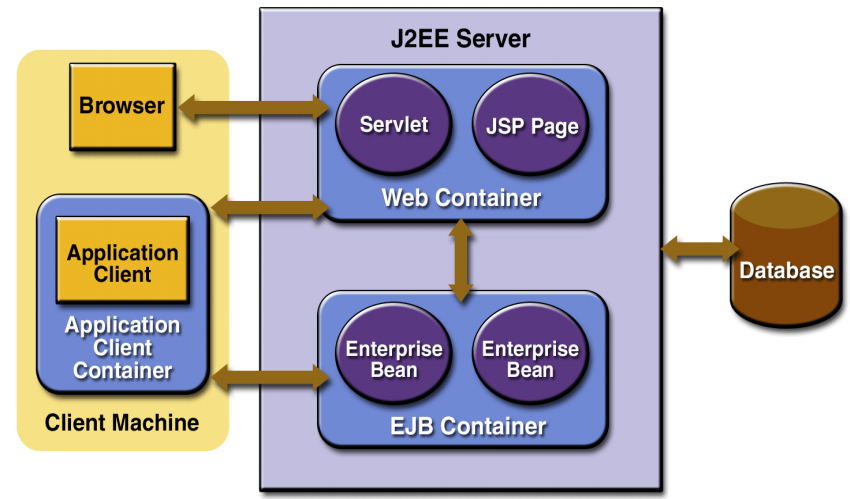
*connection.close();*

*}*

Сервер приложений

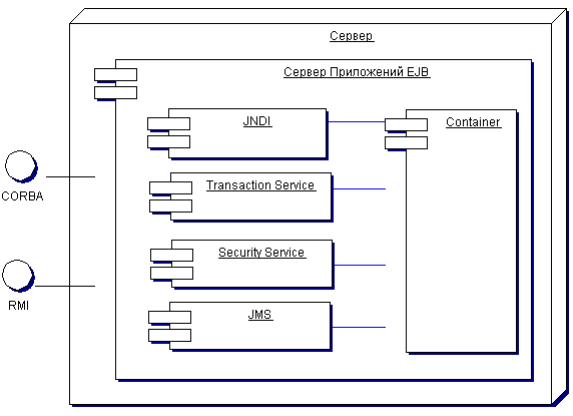
* Программная платформа для функционирования процедур(программ)
* Представляется для разработчика как большой набор готовых стандартных компонентов
* Является контейнером для бизнес-логики приложений
* Как правило являются одним из звеньев 3-х уровневой архитектуры

3-х уровневая архитектура

**

5-и уровневая архитектура в Java

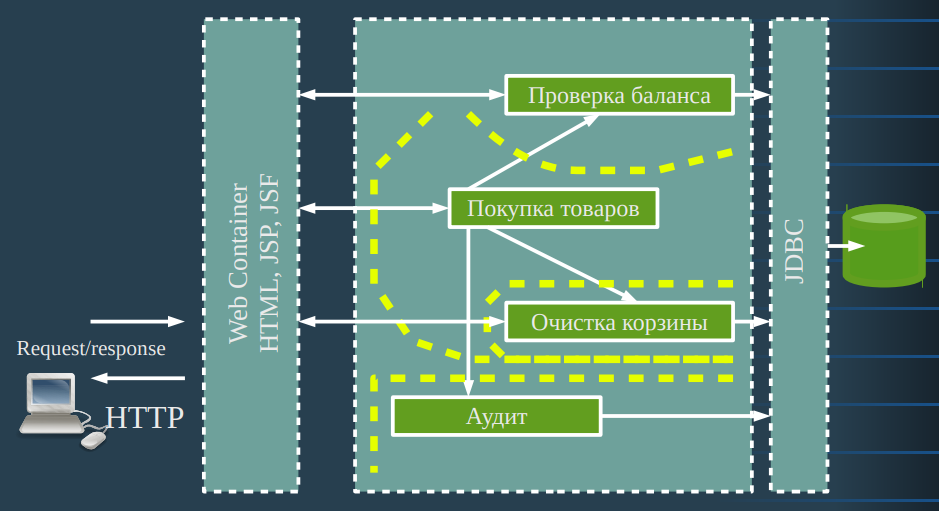
* client tier - GUI(Swing, Applet), user interaction
* presentation tier - JSP(Java Server Pages), Servlet, Session managment, Content creation
* business tier - EJB(Enterprise Java Beans), Business logic, Transaction
* integration tier - JMS(Java Message Service), JDBС(Java DataBase Connectivity), EJB, Resource adapters, Workflow
* resource tier - Databases, Data and external services

EJB(Enterprise Java Beans) контейнер

Типы EJB

* Entity Bean (Java Persistence API)
  + Отображение данных из реляционной БД на объекты
* Java Session Bean (Stateless, Statefull)
  + Содержат бизнес-логику обработки данных
  + Без хранения состояния, с сохранением состояния (Stateless, Statefull)
  + Может переиспользоваться между вызовами клиентов или быть привязанным к одному из клиентов
  + Состояние может сохраняться на диске Message
* Driven Bean
  + Содержит бизнес-логику обработки при поступлении сообщения в именованную очередь
  + Используется для асинхронной обработки и интеграции

Декомпозиция приложения



EJB. Управление транзакциями

* *@Stateless @TransactionManagement(TransactionManagementType.CONTAINER)   
  class ShopService {   
   …..…..   
  //Один вызов метода – одна транзакция*
* *@Statefull   
  @TransactionManagement(TransactionManagementType.BEAN)   
  class ComplexEventProcessor {   
  @Resource   
  private UserTransaction userTransaction;   
  …...   
  //Началом и завершением транзакции управляет класс*

EJB. Границы транзакций



EJB. Обработка исключенйи

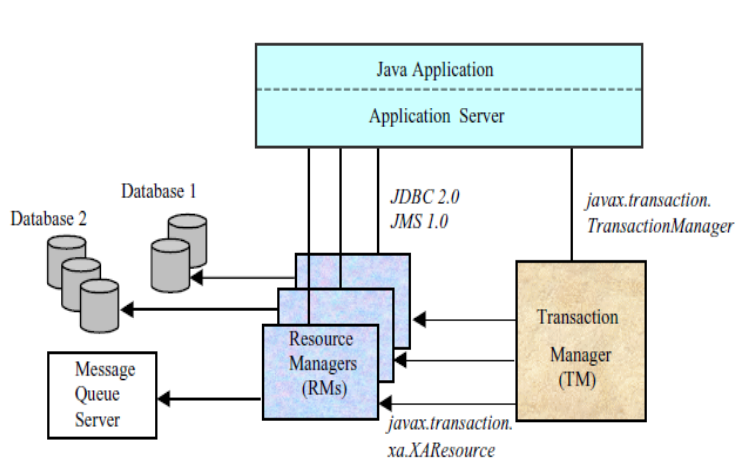
* System Exceptions – автоматический откат транзакции
  + RuntimeException, RemoteException
  + Оборачиваются в EJBException
* Application Exception
  + Checked Exception – декларируется как часть ожидаемого поведения
  + По умолчанию не приводят к откату транзакции
  + Для вызываюшей стороны приходят в неизменном виде
  + Допускается выполнение отката с использование аннотации у исключения
  + *@ApplicationException(rollback=true)   
    Class MyAppException extends Exception {   
     …...*.

1. **Управление транзакциями в приложениях. Стандарт JTA. Компоненты, участвующие в обработке транзакций и их роль**

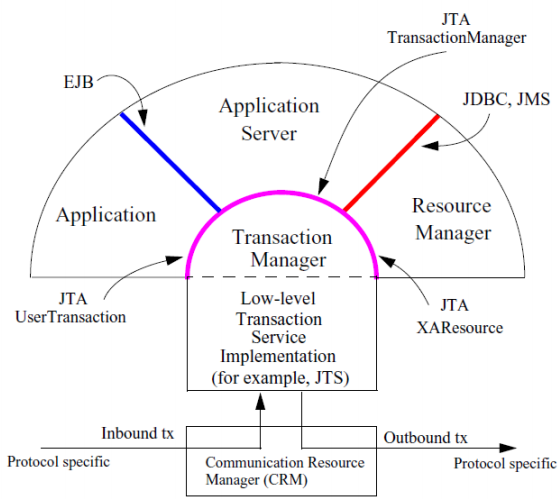
Java Transaction API (JTA 1.1)

* Локальные транзакции
  + Работаем из EJB через JDBC/JMS/специфический протокол с ресурсами напрямую выполняя COMMIT/ROLLBACK
  + Принимаем самостоятельно решение о COMMIT/ROLLBACK
* Распределенные транзакции
  + Вовлечено много ресурсов в процессе обработки бизнес-логики
  + Единый для всех интерфейс для взаимодействия с текущей транзакцией (JTA) UserTransaction
  + Решение о COMMIT/ROLLBACK может приниматься контейнером или классом
  + Все ресурсы должны поддерживать 2-х фазный протокол (кроме одного)(XA-протокол)

Распределенные транзакции



Взаимодействие компонентов при обработке транзакции



Интерфейс TransactionManager

Интерфейс между сервером приложений и менеджером транзакций

*interface javax.transaction.TransactionManager {*

*public void begin();*

*public void commit();*

*public int getStatus();*

*public Transaction getTransaction();*

*public void resume(Transaction tobj);*

*public void rollback();*

*public void setRollbackOnly();*

*public void setTransactionTimeout(int seconds);*

*public Transaction suspend() ;*

*}*

Интерфейс Transaction

Интерфейс управления транзакцией и ее RM для сервера приложений

*interface javax.transaction.Transaction {*

*public void commit();*

*public boolean delistResource(XAResource xaRes, int flag);*

*public boolean enlistResource(XAResource xaRes);*

*public int getStatus();*

*public void registerSynchronization(Synchronization sync);*

*public void rollback();*

*public void setRollbackOnly();*

*}*

Интерфейс UserTransaction

Интерфейс управления транзакцией для приложения public

*interface javax.transaction.UserTransaction {*

*public void begin();*

*public void commit();*

*public int getStatus();*

*public void rollback();*

*public void setRollbackOnly();*

*public void setTransactionTimeout(int seconds);*

*}*

Интерфейс Xid

Интерфейс доступа к идентификатору транзакции

*public interface javax.transaction.xa.Xid {*

*int getFormatId();*

*byte[] getGlobalTransactionId();*

*byte[] getBranchQualifier();*

*}*

Интерфейс XAResource

Транзакционный интерфейс доступа к ресурсам (RM)

*public interface javax.transaction.xa.XAResource {*

*public void commit(Xid xid, boolean onePhase);*

*public void end(Xid xid, int flags);*

*public void forget(Xid xid);*

*public int getTransactionTimeout();*

*public boolean isSameRM(XAResource xares);*

*public int prepare(Xid xid);*

*public Xid[] recover(int flag);*

*public void rollback(Xid xid);*

*public boolean setTransactionTimeout(int seconds);*

*public void start(Xid xid, int flags);*

*}*

Немного о JNDI

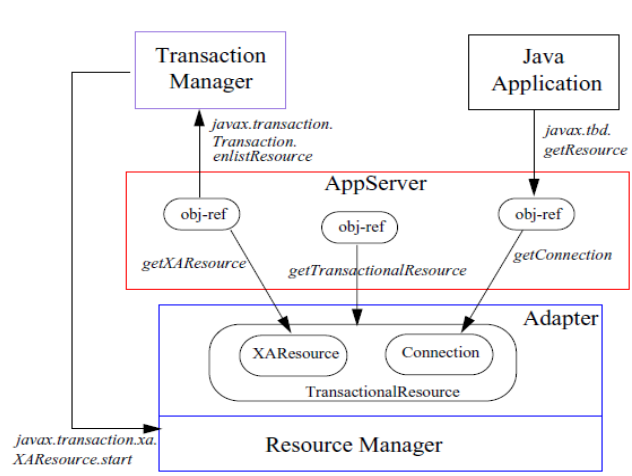
* Java Naming Directory Interface
* Содержит именованные ссылки на объекты и фабрики объектов
  + java:comp/TransactionManager
  + java:comp/UserTransaction

1. **Управление транзакциями в приложениях. Стандарт JTA. Процедура получения и работы с транзакционным ресурсом.**

Java Transaction API (JTA 1.1)

* Локальные транзакции
  + Работаем из EJB через JDBC/JMS/специфический протокол с ресурсами напрямую выполняя COMMIT/ROLLBACK
  + Принимаем самостоятельно решение о COMMIT/ROLLBACK
* Распределенные транзакции
  + Вовлечено много ресурсов в процессе обработки бизнес-логики
  + Единый для всех интерфейс для взаимодействия с текущей транзакцией (JTA) UserTransaction
  + Решение о COMMIT/ROLLBACK может приниматься контейнером или классом
  + Все ресурсы должны поддерживать 2-х фазный протокол (кроме одного)(XA-протокол)

Получение ресурсов через сервер приложений



Получение ресурса сервером приложений

*// Запрос от приложения на получение ресурса/соединения*

*Context ctx = InitialContext();*

*ResourceFactory rf =(ResourceFactory)ctx.lookup(“MyEISResource”); TransactionalResource res = rf.getTransactionalResource();*

*// Получение XAResource части ресурса для менеджера транзакций XAResource xaRes = res.getXAResource(); (TransactionManager.getTransaction()).enlistResource(xaRes);*

*// Получение соединения как части ресурса*

*Connection con = (Connection)res.getConnection();*

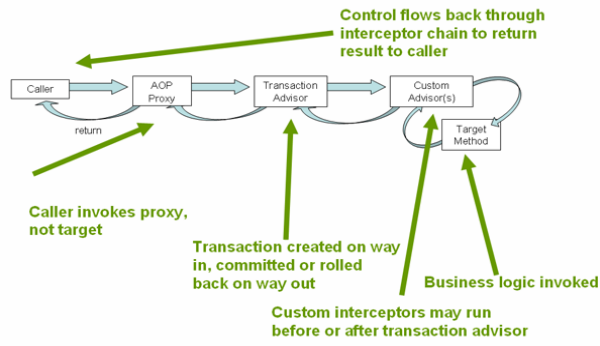
*return con;*

**

1. **Управление транзакциями в приложениях. Использование AOP в SpringFramework для управления транзакциями**

Управление транзакциями в SpringFramework

* Aspects Oriented Programming (AOP)
* Injection-of-Control (IoC)



Пример сервиса

*interface BookService {*

*void buyBooks(....);*

*}*

*class DefaultBookService {*

*void buyBooks(....);*

*}*

*class MyPage {*

*BookService bookService; {*

*….*

*bookService.buyBooks(....)*

*}*

*}*

Описание транзакционного сервиса

