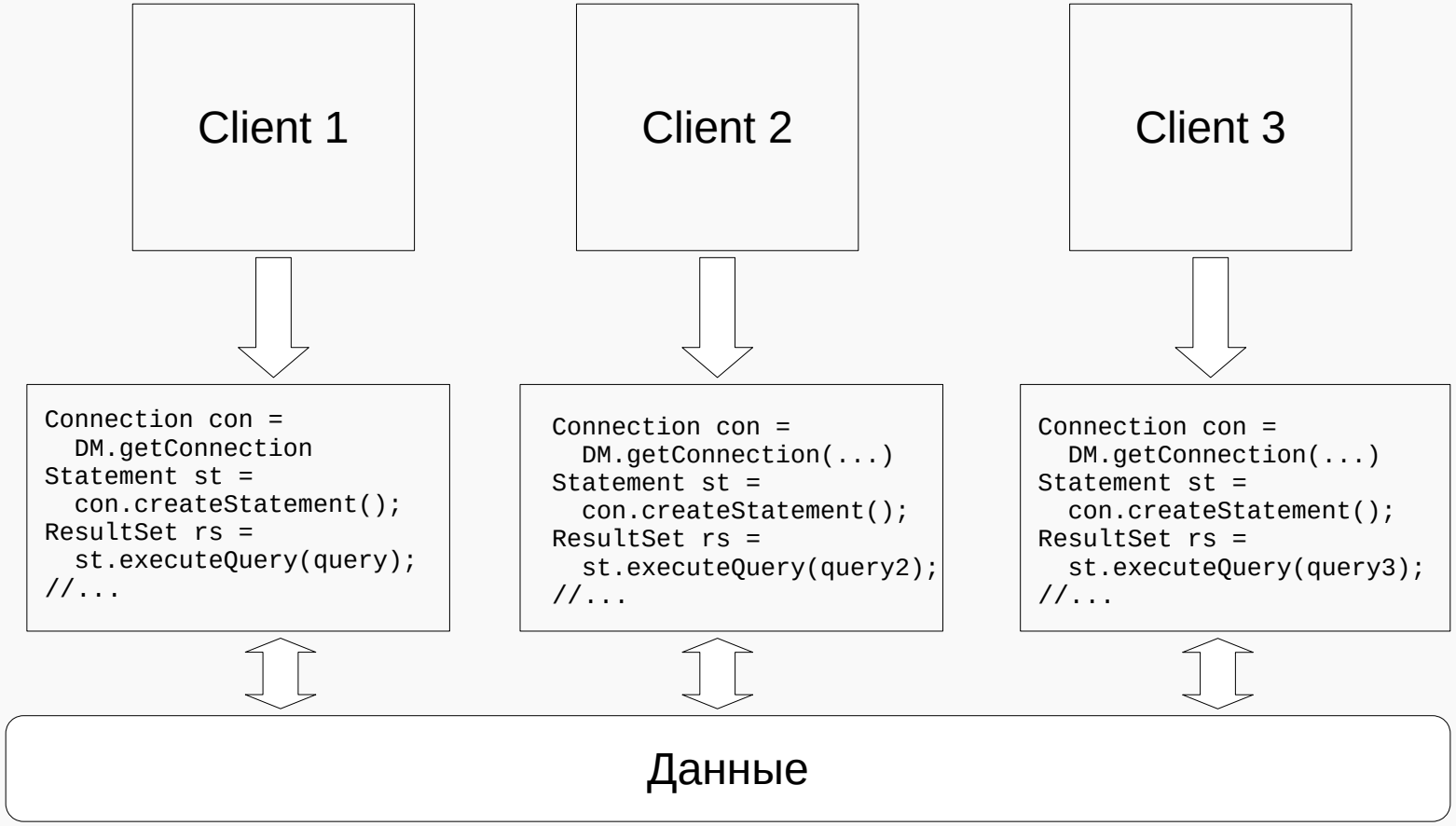
1. Особенности взаимодействия с БД

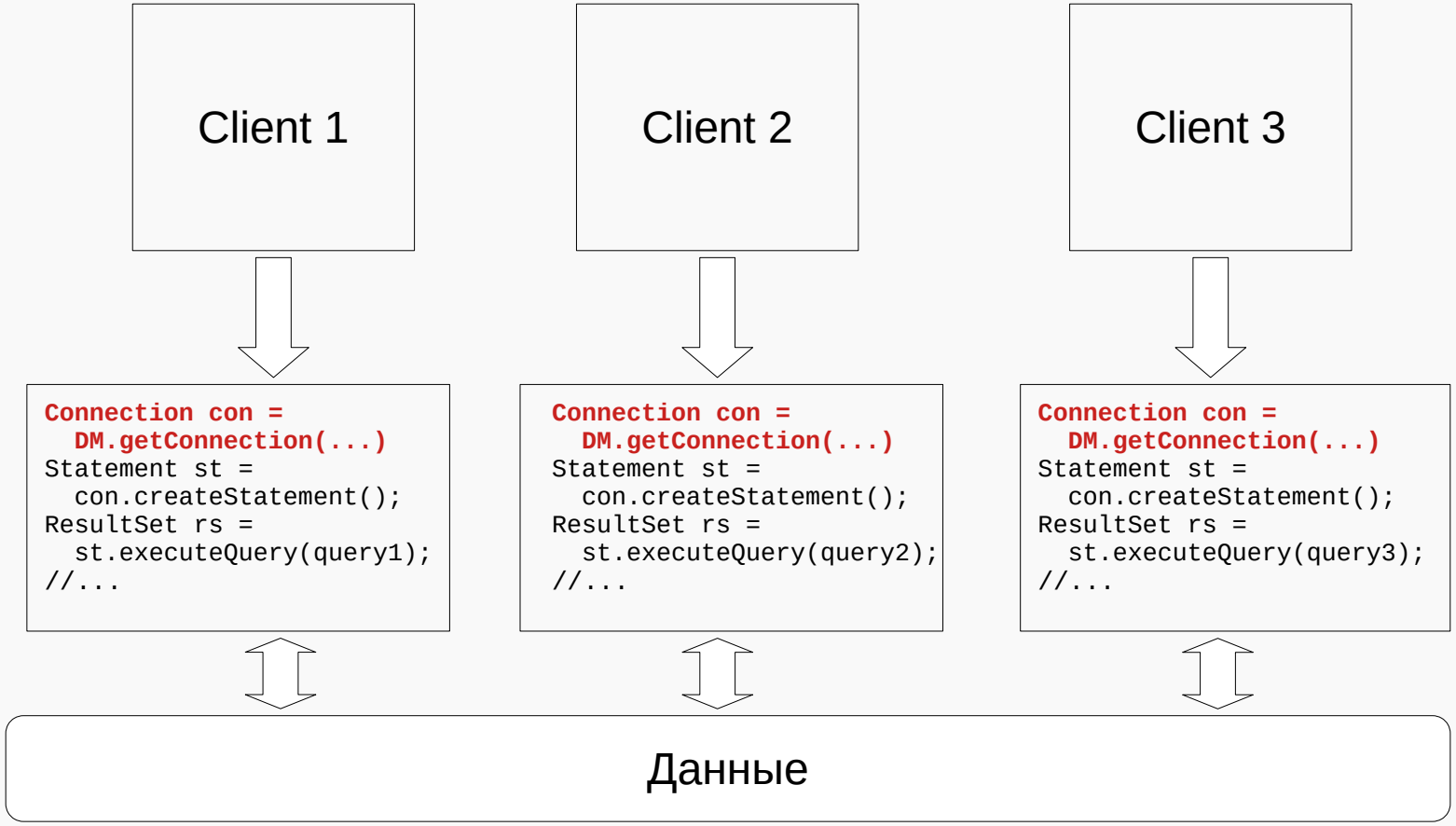
JDBC

Что можно улучшить?



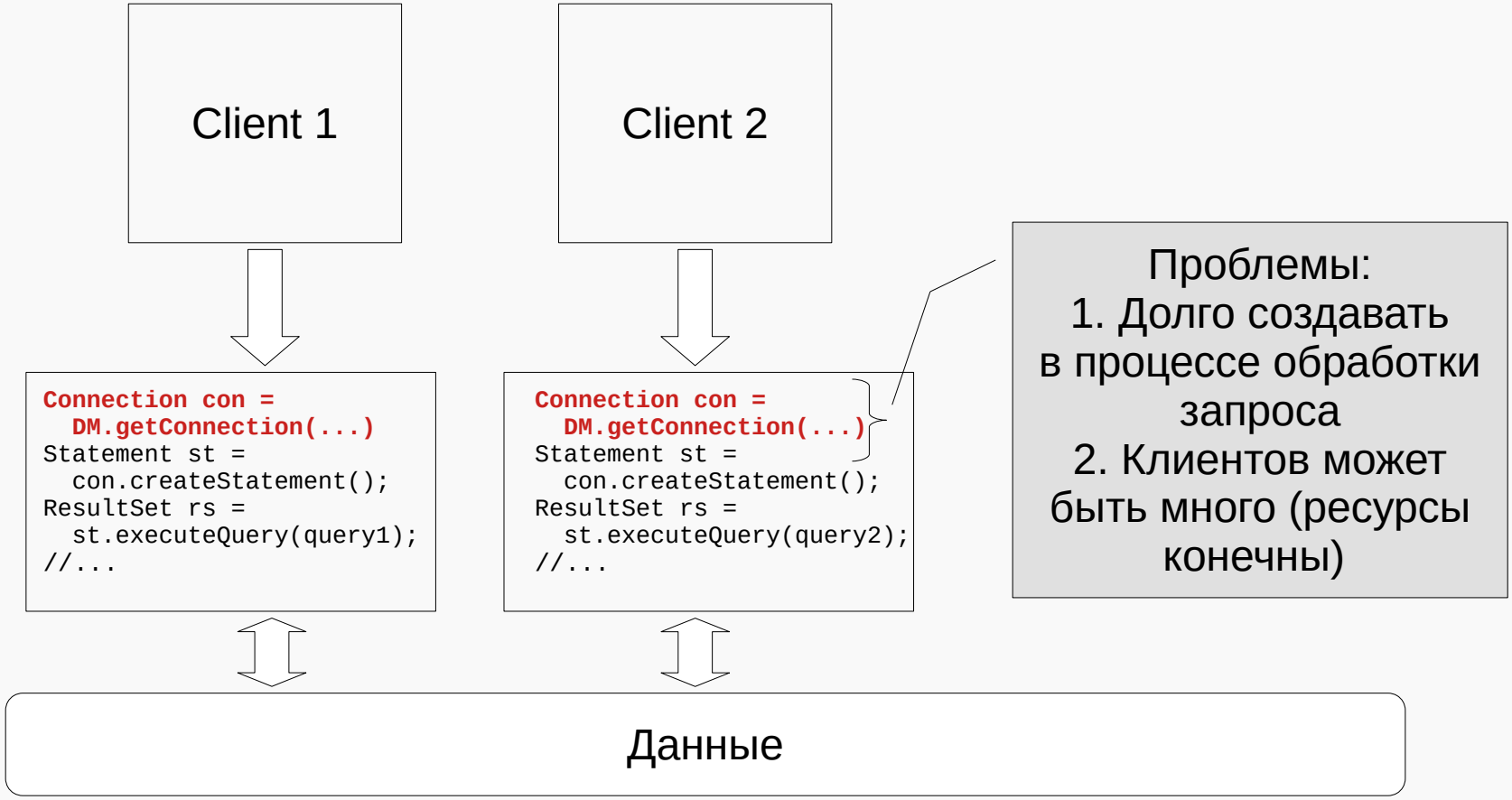
JDBC

У каждого подьзователя — свое подключение (создается в момент вызова).



JDBC

У каждого подьзователя — свое подключение (создается в момент вызова).



ИТМО ВТ

Connection Pool (1)

Connection Pool:

- Осуществление подключения к БД — дорогая операция.
- Решение: сделать «пул» активных подключений, которые можно использовать для подключения к БД.
- Есть разные реализации: HikariCP, C3Po, Apache DBCP.

Connection Pool (3)

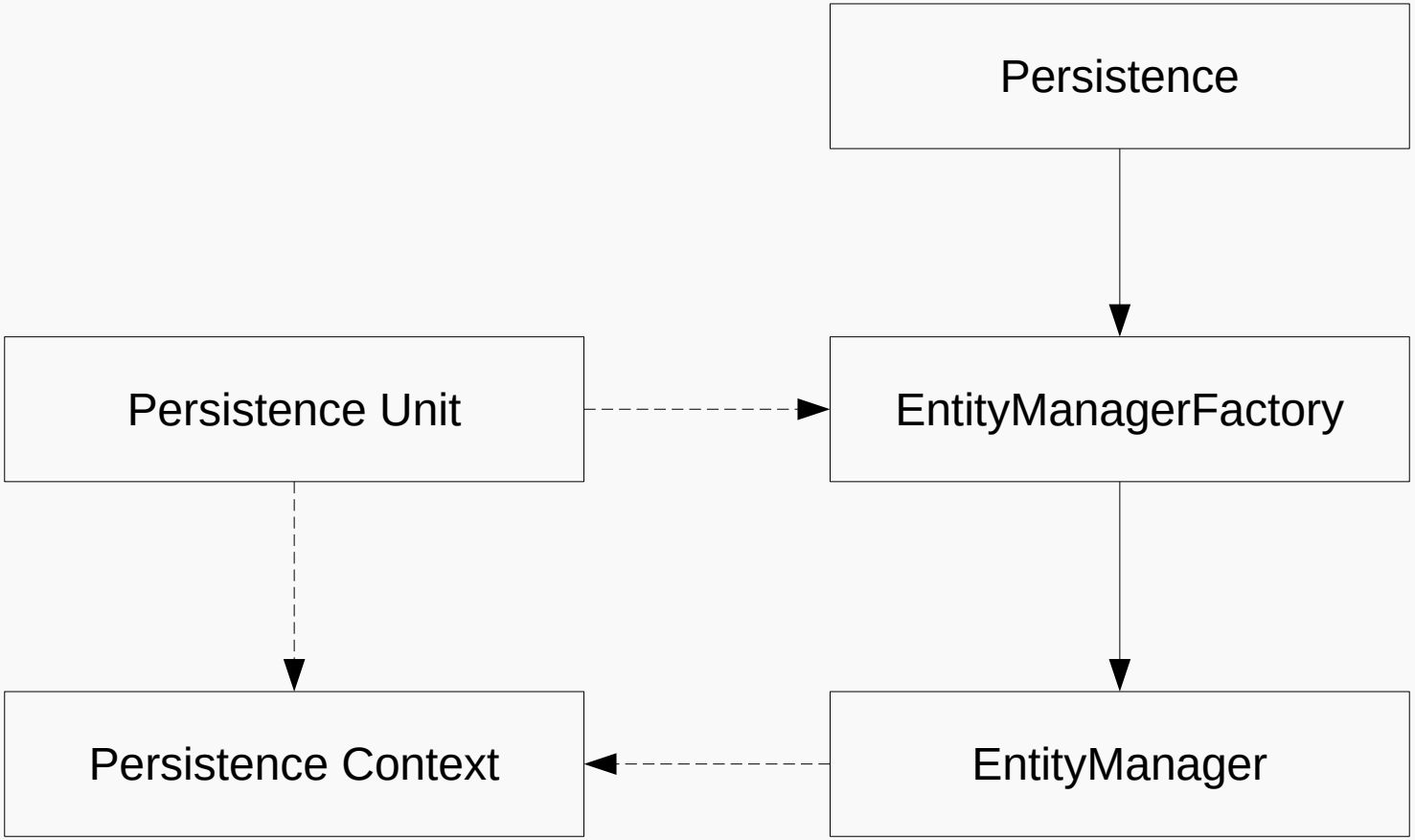
```
InitialContext ctx = new InitialContext();
javax.sql.DataSource dataSource =
    (javax.sql.DataSource)ctx.lookup("jdbc/StudPool");
Connection connection = dataSource.getConnection();
String studQuery = "select * from STUDENTS";
Statement st = con.createStatement();
ResultSet rs = st.executeQuery(studQuery);
st.close();
```

Что изменилось:
1. Получаем StudPool (jndi)
2. Из StudPool получаем подключение

2. Транзакции и JTA

ИТМО ВТ

JPA


Persistence
Persistence Unit
EntityManagerFactory
Persistence Context
EntityManager

ИТМО ВТ

Базовые классы и интерфейсы

jakarta.persistence.Persistence

- Корневой класс для получения Entity Manager (Java SE environment)
- Определяет Persistence Provider для данного Persistence Unit
- Позволяет получить EntityManagerFactory

jakarta.persistence.EntityManagerFactory

- Отвечает за создание EntityManagers для заданного Persistence Unit или конфигурации

ИТМО ВТ

Persistence Unit

Persistence Unit содержит:

- Информацию о Persistence Context;
- Настройки источника данных;

Persistence Unit связан с одной EntityManagerFactory и всеми EntityManagers, созданными ей;

ИТМО ВТ

Persistence Context

- Persistence Context – абстрактное представление множества управляемых объектов-сущностей;
- PC контроллируется и управляется EntityManager;
- Содержимое PC изменяется в результате операций, производимых EntityManager API;
- PC может принадлежать нескольким EntityManager;

ИТМО ВТ

EntityManager

Базовый интерфейс для работы с хранимыми данными:

- Обеспечивает взаимодействие с Persistence Context;
- Можно получить через EntityManagerFactory.
- Обеспечивает базовые операции для работы с данными (CRUD).

ИТМО ВТ

EntityManager

EntityManager:

- Container-managed EntityManager.
- Application-managed EntityManager:
  - приложение само получает и закрывает PC, когда необходимо.
  - Application-managed API может быть использовано как в Java EE, так и в Java SE приложениях.

ИТМО ВТ

EntityManager (app-managed)

```
public class EMTest {
    public static void main(String[] args) {
        EntityManagerFactory emFactory =
            Persistence
                .createEntityManagerFactory("studentPU");
        EntityManager entityManager =
                emFactory.createEntityManager();
        entityManager.getTransaction().begin();
        //операции над сущностями
        entityManager.getTransactioin().commit();
        entityManager.close(); emFactory.close();
    }
}
```

ИТМО ВТ

Транзакции и JPA

- RESOURCE-LOCAL
- JTA Transactions

ИТМО ВТ

RESOURCE LOCAL

- В Java SE или Java EE (Application Managed).
- Транзакции управляются через EntityTransaction:
  - begin
  - commit
  - rollback

RESOURCE LOCAL (1)

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
persistence_1_0.xsd" version="1.0">
   <persistence-unit name="studPU" transaction-type="RESOURCE_LOCAL">
      <non-jta-data-source>studSource</non-jta-data-source>
   </persistence-unit>
</persistence>
```

При конфигурации через RESOURCE_LOCAL и DataSource нужно исп-ть non-jta-data-source

RESOURCE LOCAL (2)

```
public class EMTest {
    public static void main(String[] args) {
        EntityManagerFactory emFactory =
            Persistence
                .createEntityManagerFactory("studPU");
        EntityManager entityManager =
                emFactory.createEntityManager();
        entityManager.getTransaction().begin();
        //операции над сущностями
        entityManager.getTransactioin().commit();
        entityManager.close(); emFactory.close();
    }
}
```

ИТМО ВТ

Java Transaction API

- API для управления транзакциями (в том числе распределенными).
- Специфицирован, входит в состав Java EE.
- Контейнер содержит менеджер транзакций, манипулирующий ресурсами.
- Реализует стандарт X/OpenXA.

ИТМО ВТ

Транзакции в современных ИС

- “Классическая” модель реализации на уровне СУБД часто не хватает:
  - Распределённый и гетерогенный уровень хранения – разные БД на разных узлах и т.д.
  - Распределённая и гетерогенная архитектура на уровне бизнес-логики.
- Возможные решения:
  - Не использовать “глобальные” транзакции.
  - Управлять транзакциями со стороны бизнес-логики.

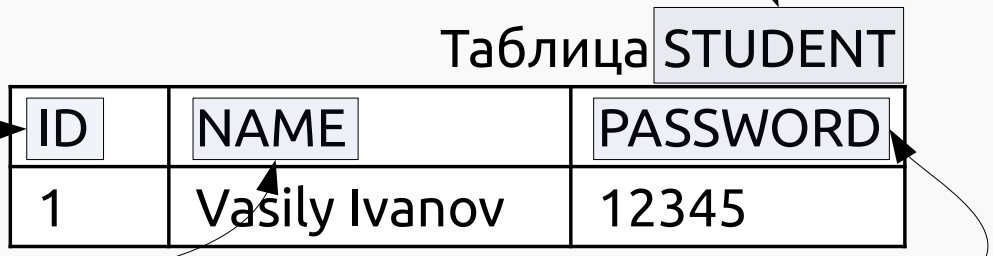
JTA, persistence.xml

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
persistence_1_0.xsd" version="1.0">
   <persistence-unit name="studPU" transaction-type="JTA">
      <jta-data-source>studSource</jta-data-source>
   </persistence-unit>
</persistence>
```

При конфигурации через
JTA и DataSource
нужно исп-ть jta-data-source

Student Entity

```
@Entity
public class Student {
    @Id
    int id;

    String name;
    String password;
}
```



JTA transactions

```
@Stateless
public class StudentEJB {
   @PersistenceContext
    private EntityManager em;
    @TransactionAttribute(
            TransactionAttributeType.REQUIRED)
    public void updateStudentName(
       Integer studId, String name) {
       try {
           Student stud = em.find(Student.class, studId);
           stud.setName(name);
           em.persist(stud);
        } catch ( … //catch finally implementation ) { }
    }
}
```

В транзакции

ИТМО ВТ

Особенности JTA

Два высокоуровневых режима управления транзакциями (в EJB):

- программный (BMT)
- декларативный (CMT).

ИТМО ВТ

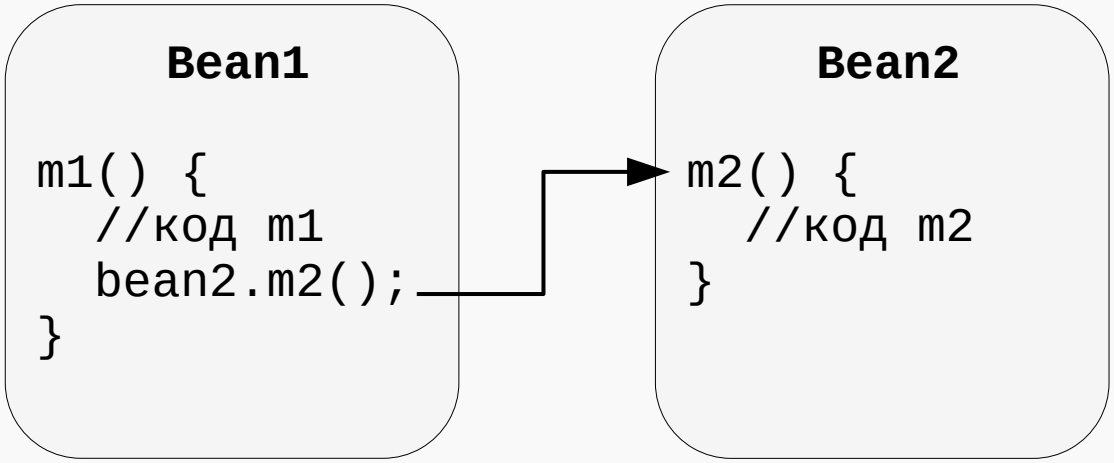
JTA: CMT

- Решения об открытии и закреплении транзакции принимает контейнер.
- Границы транзакции определяются границами методов и используемыми аннотациями.
- Что делать при вызовах методов друг из друга, также определяется аннотациями.

JTA: CMT

```
@Stateless
public class StudentEJB {
   @PersistenceContext
    private EntityManager em;
        @TransactionAttribute(
            TransactionAttributeType.REQUIRED)
       public void updateStudentName(
                                Integer studId, String name) {
           try {
               Student stud = em.find(Student.class, studId);
               stud.setName(name);
               em.persist(stud);
           } catch ( … //catch finally implementation ) { }
    }
}
```

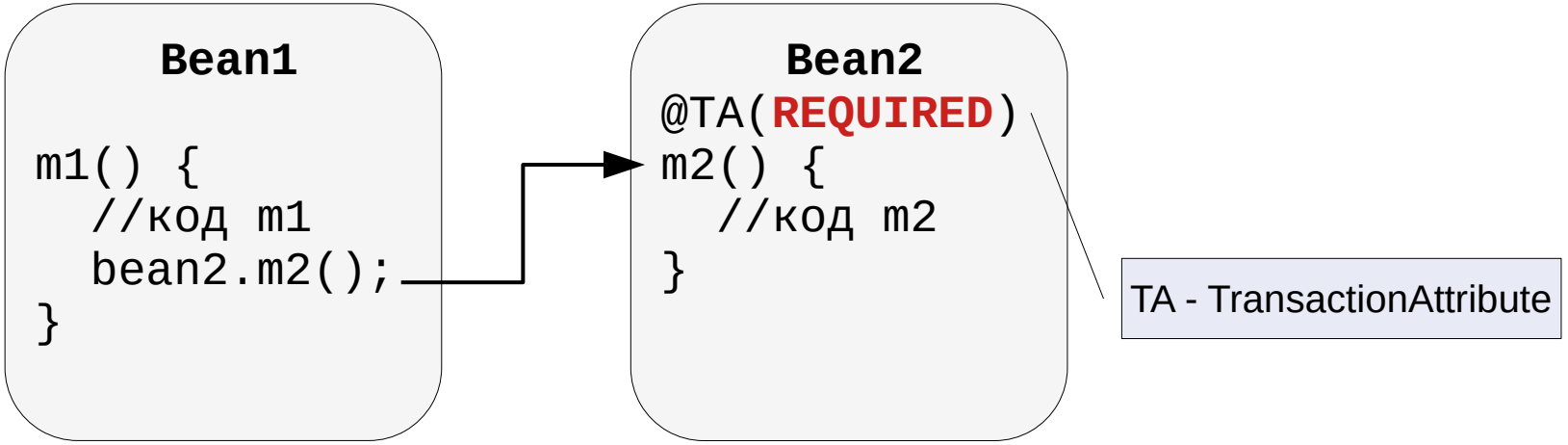
CMT: @TransactionAttribute



- Есть 6 способов включения методов в транзакции друг друга.
- Они определяются значением атрибута @TransactionAttribute над классом или методом.
- Значение атрибута @TransactionAttribute над методом переопределяет значение “по умолчанию”, взятое из класса.

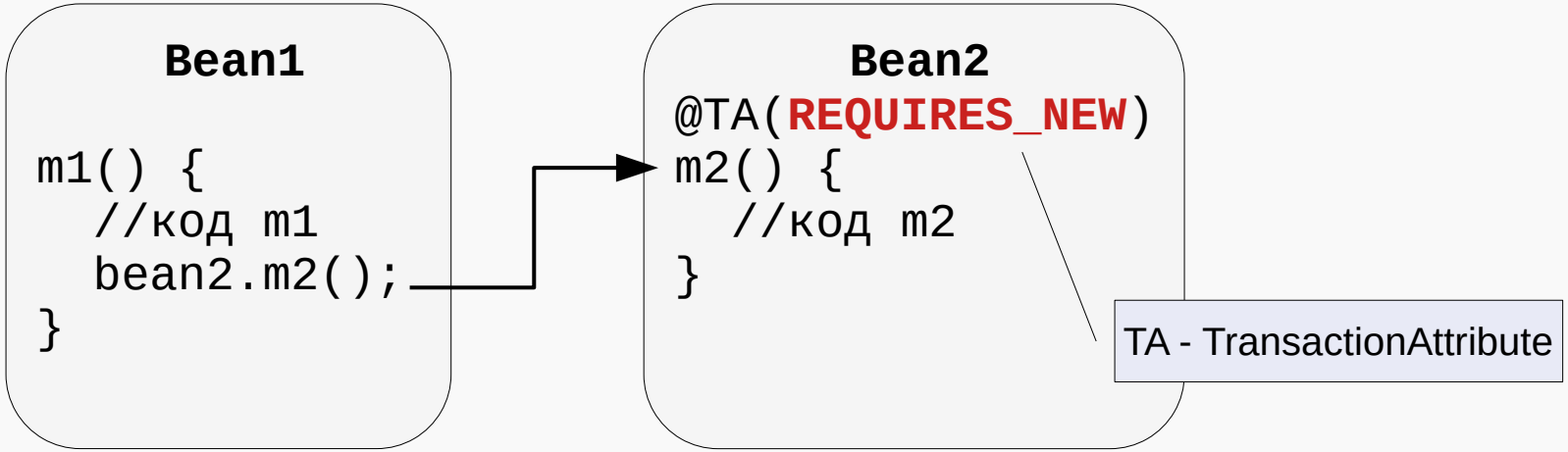
@TransactionAttribute (Required)

- Required – метод требует выполнения в рамках транзакции.
- Если вызывающий метод связан с транзакцией — происходит “включение” вызываемого метода в контекст транзакции вызывающего. Если вызывающий метод не в контексте транзакции — создается новая.



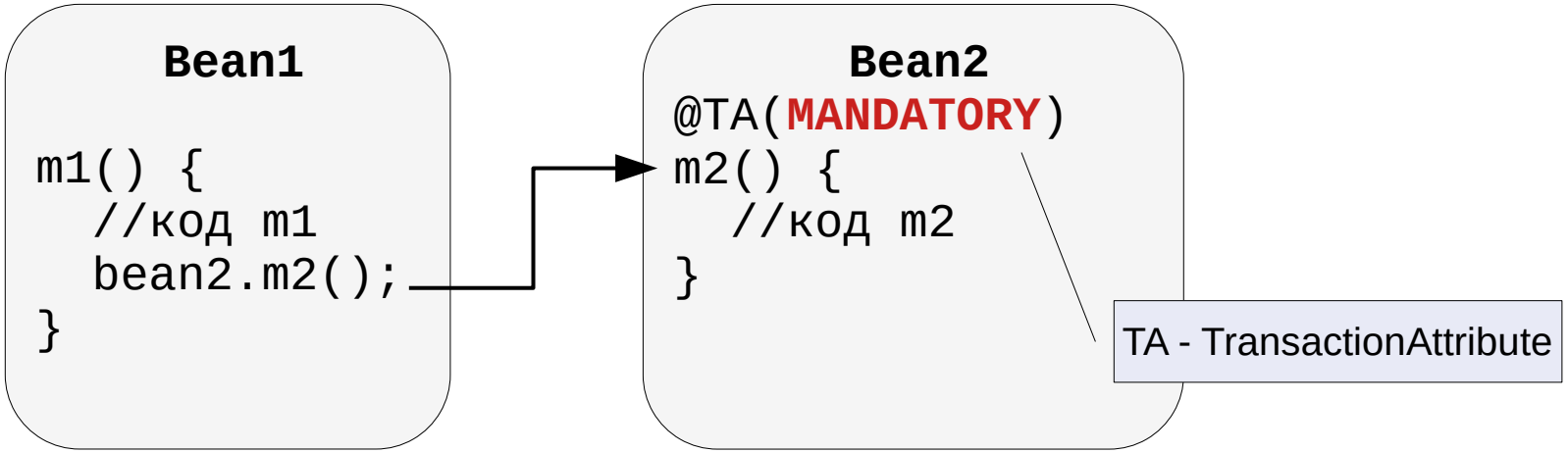
@TransactionAttribute (Requires_New)

- Requires_New – всегда создаёт новую транзакцию.
- Если есть существующая (в вызываемом методе) – она приостанавливается до завершения выполнения новой.



@TransactionAttribute (Mandatory)

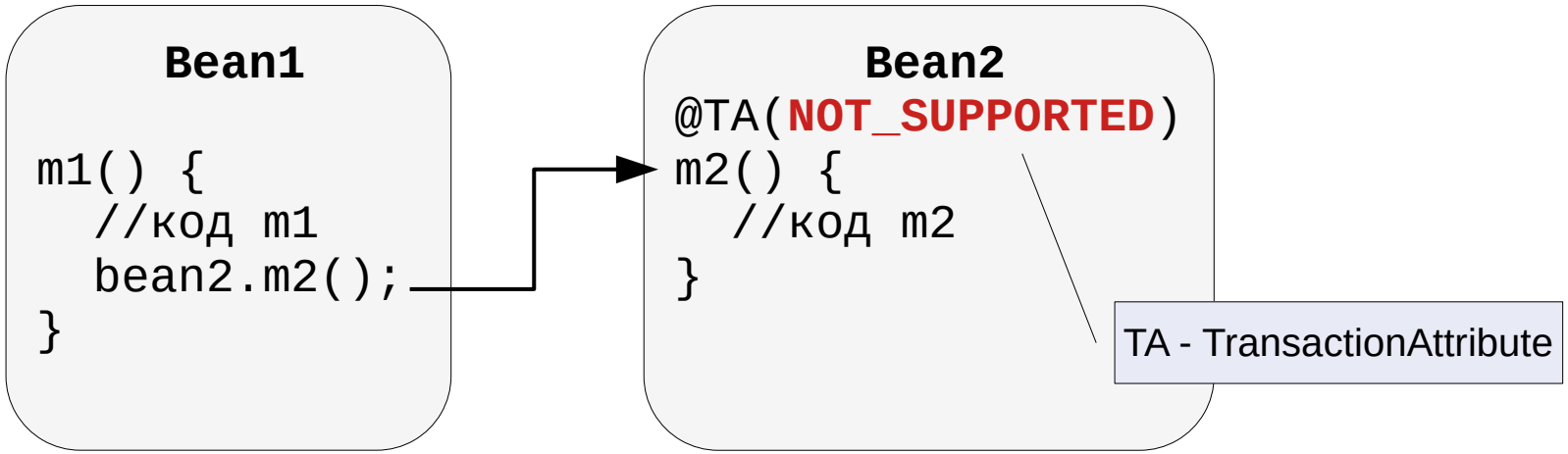
- Mandatory – метод требует выполнения в рамках транзакции.
- Если вызывающий метод связан с транзакцией — происходит “включение” вызываемого метода в контекст транзакции вызывающего. Если вызывающий метод не в контексте транзакции — TransactionRequiredException



30

@TransactionAttribute (Not_Supported)

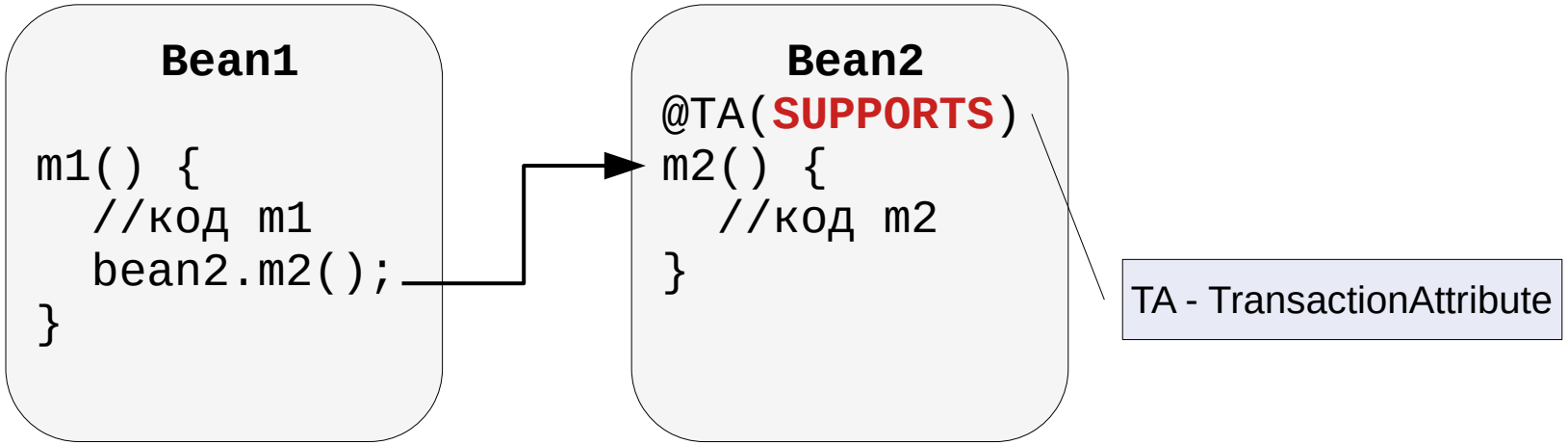
- NotSupported – не использует транзакцию.
- Если клиент связан с транзакцией, приостанавливает текущую транзакцию до возврата из метода.



31

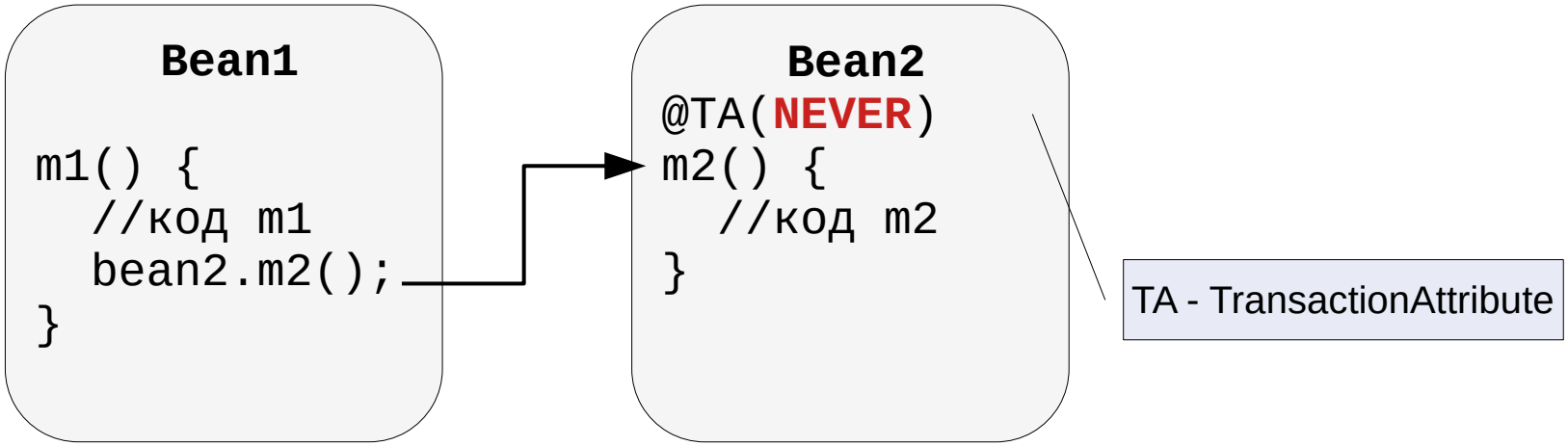
@TransactionAttribute (Supports)

- Supports – если текущая транзакция есть (в вызывающем методе) – “включает” в неё вызываемый метод.
- Если её нет – не требует транзакции.



@TransactionAttribute (Never)

- Never – если текущая транзакция есть – выбрасывает RemoteException. Выполняется, если нет транзакции.



33

ИТМО ВТ

Пример использования CMT

```
@TransactionAttribute(NOT_SUPPORTED)
@Stateful
public class MyBean {
  @TransactionAttribute(REQUIRED)
  public void myTest1() { /* impl */ }

  @TransactionAttribute(REQUIRES_NEW)
  public void myTest2() { /* impl */ }

  public void myTest3() { /* impl */ }
}
```

ИТМО ВТ

JTA: BMT

- Где открывается и закрепляется транзакция определяется пользователем в бине:

```
public void testMethod() {
    //code before transaction
    begin transaction
    ...
    update table1
    ...
    if (условие 1)
        commit transaction
    else if (условие 2)
        rollback transaction
    else
        rollback transaction
        begin transaction
        update table3
        commit transaction
    //code after transaction
}
```

ИТМО ВТ

JTA: BMT

```
@Stateless
@TransactionManagement(
    TransactionManagementType.BEAN)
public class StudentEJB {
    @PersistenceContext
    private EntityManager em;
    @Inject
    private UserTransaction ut;

    public void updateStudentName(
                            Integer studId, String name) {
        try {
            ut.begin();
            //impl here
            ut.commit();
        } catch (Exception e) { /* exc processing */ }
    }
}
```

JTA: BMT

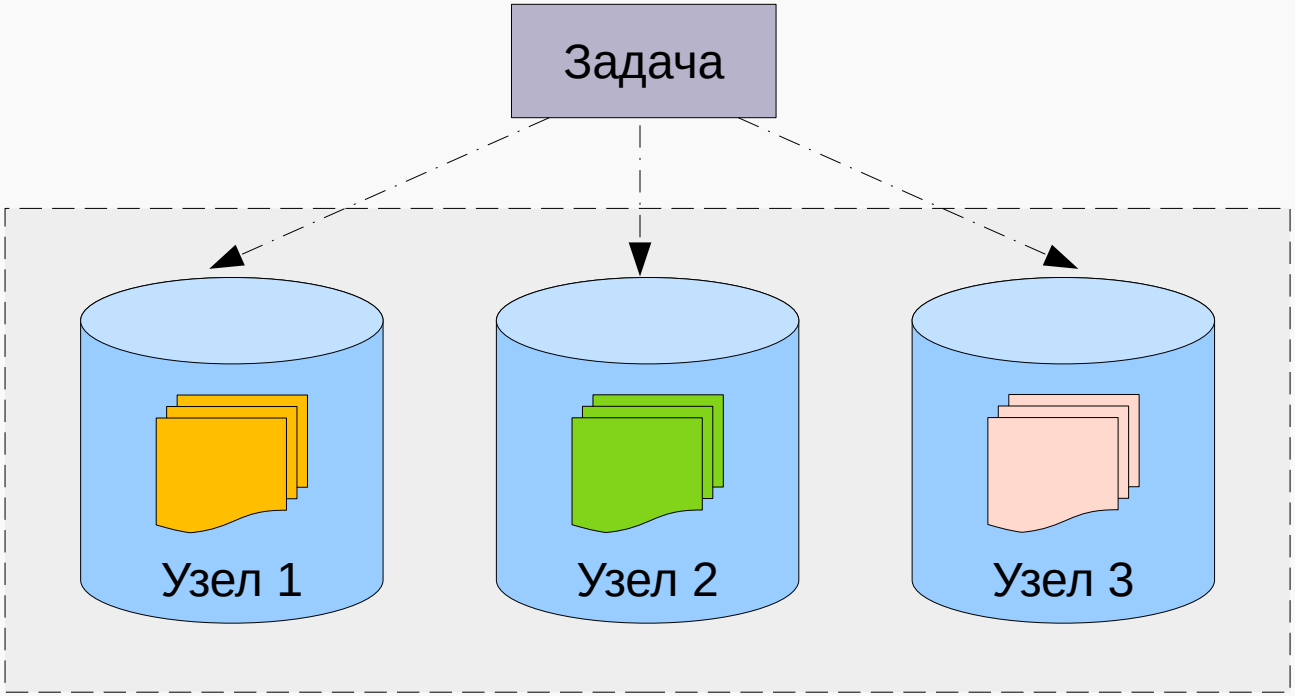
```
    public void updateStudentName(Integer studId, String name) {
        try {
            ut.begin();
            Student stud = em.find(Student.class, studId);
            stud.setName(name);
            em.persist(stud);
            ut.commit();
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            try {
                if (ut.getStatus() == Status.STATUS_ACTIVE)
                    ut.rollback();
            } catch (Throwable e) { /* some impl */}
        }
    }
}
```

3. Работа с распределенными ресурсами

ИТМО ВТ

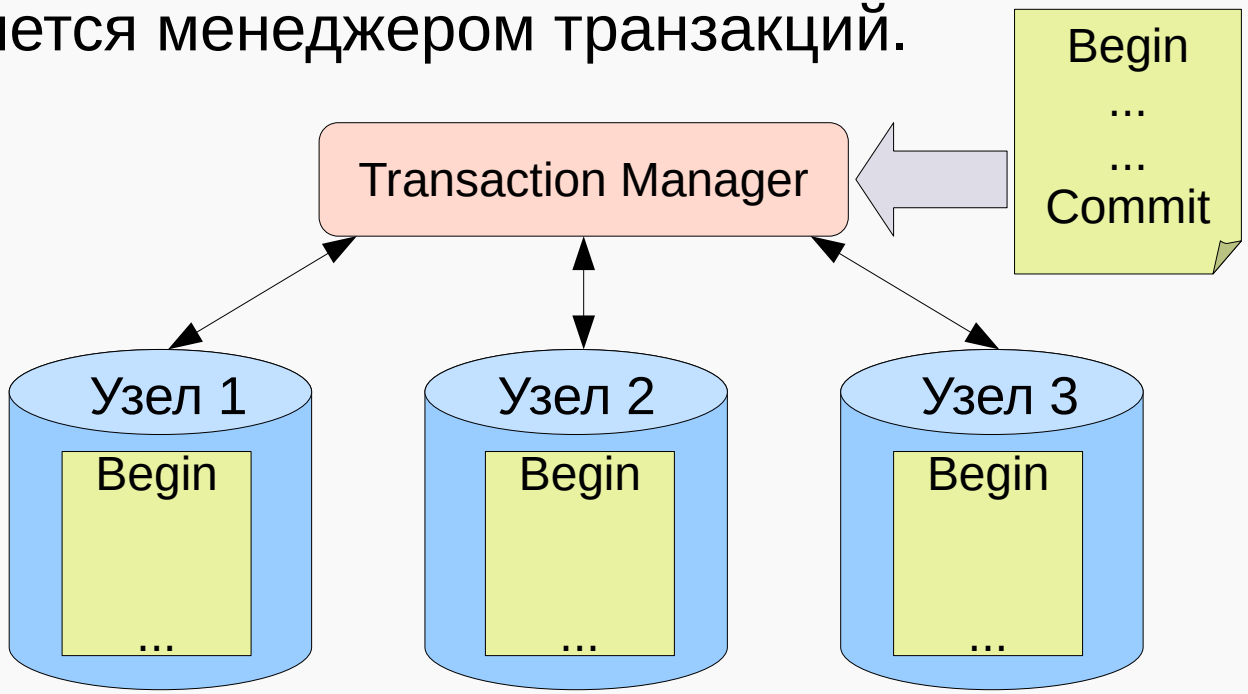
Распределенные данные

- Данные могут быть расположены (распределены) на нескольких узлах.
  - Узел ~ в составе одной базы данных (одна/разные) машины; разные БД.



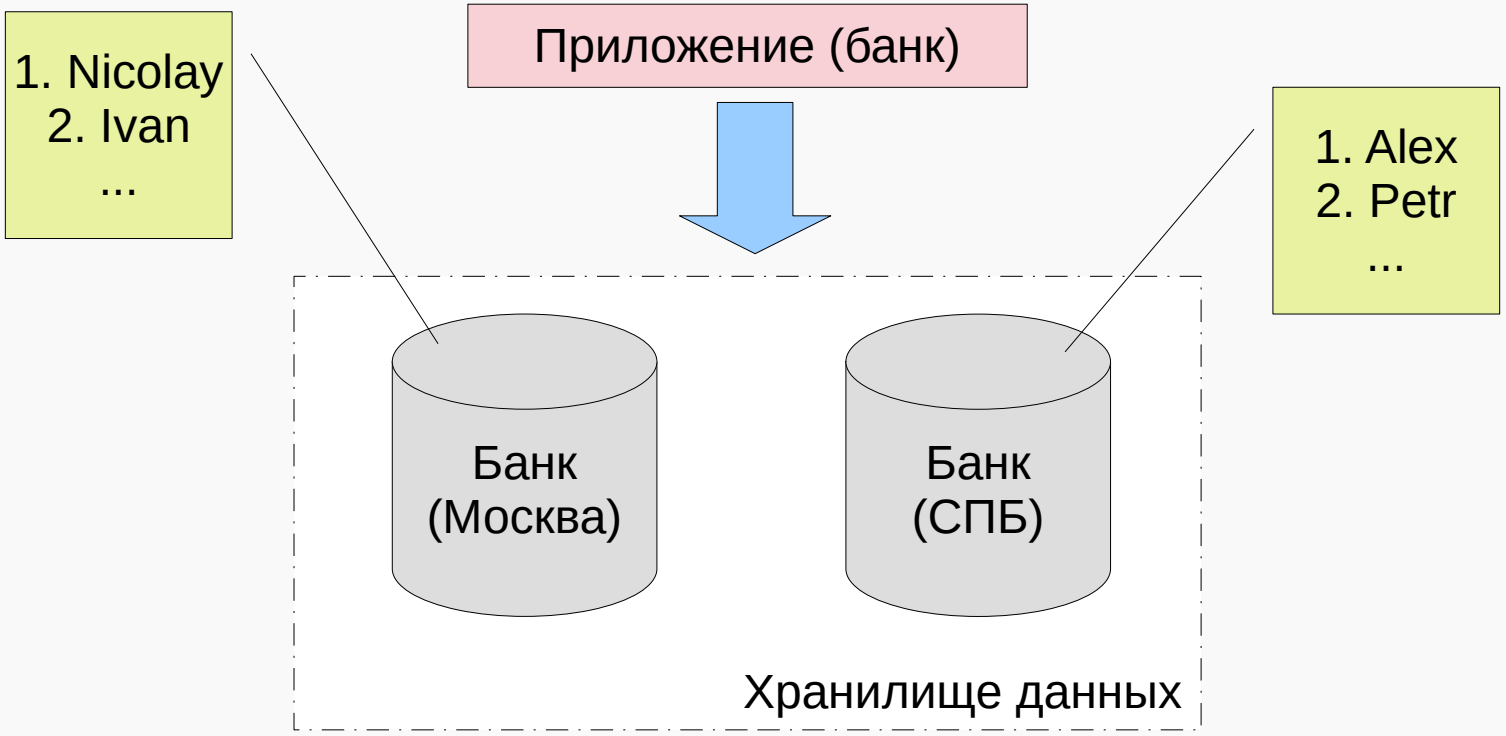
Распределенные транзакции

- Требуется обеспечить атомарность при выполнении транзакций над такими распределенными данными.
- Распределенная транзакция — набор заданий на разных узлах;
  - управляется менеджером транзакций.



ИТМО ВТ

Пример распределенной системы



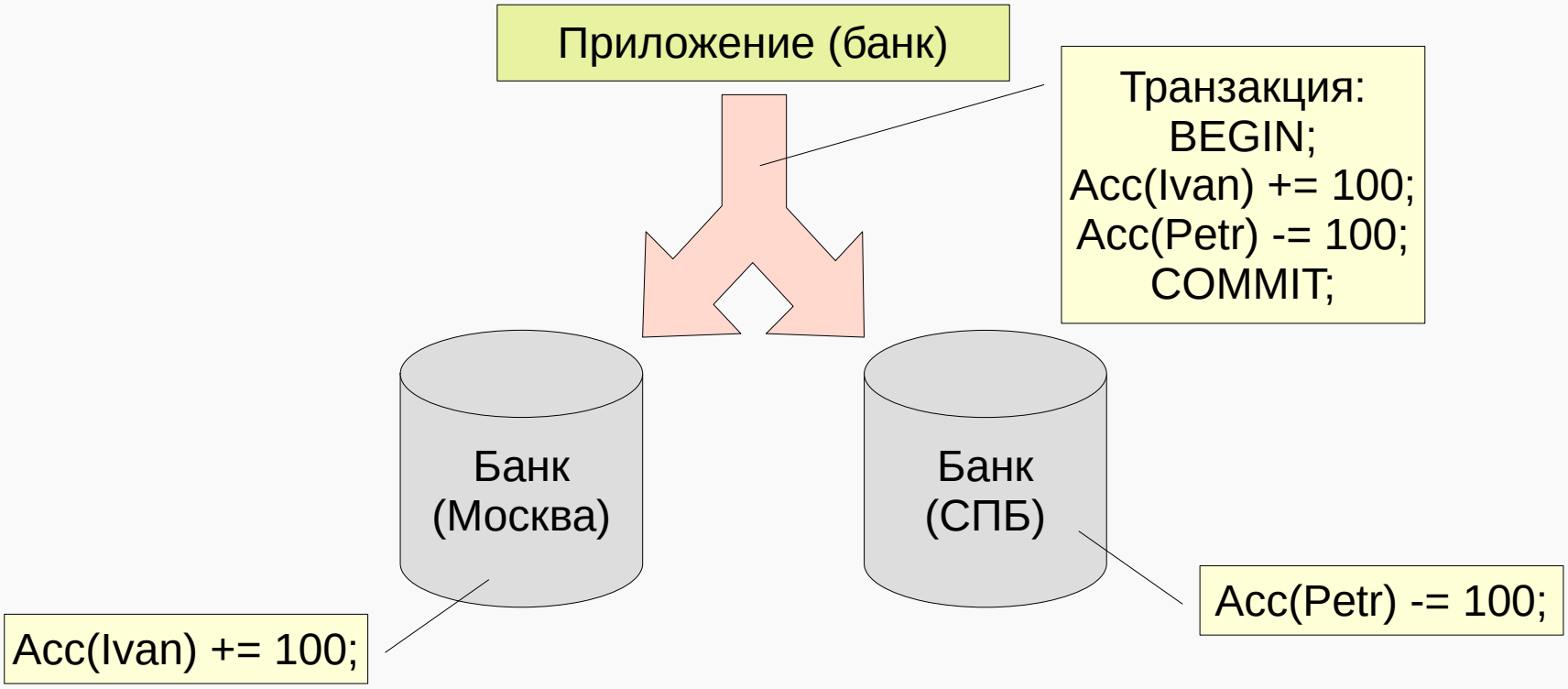
ИТМО ВТ

Особенности при работе с распределенными транзакциями

- Как обеспечить последовательное исполнение операций в транзакции?
  - глобальные блокировки.
- Проблема фиксации транзакций.

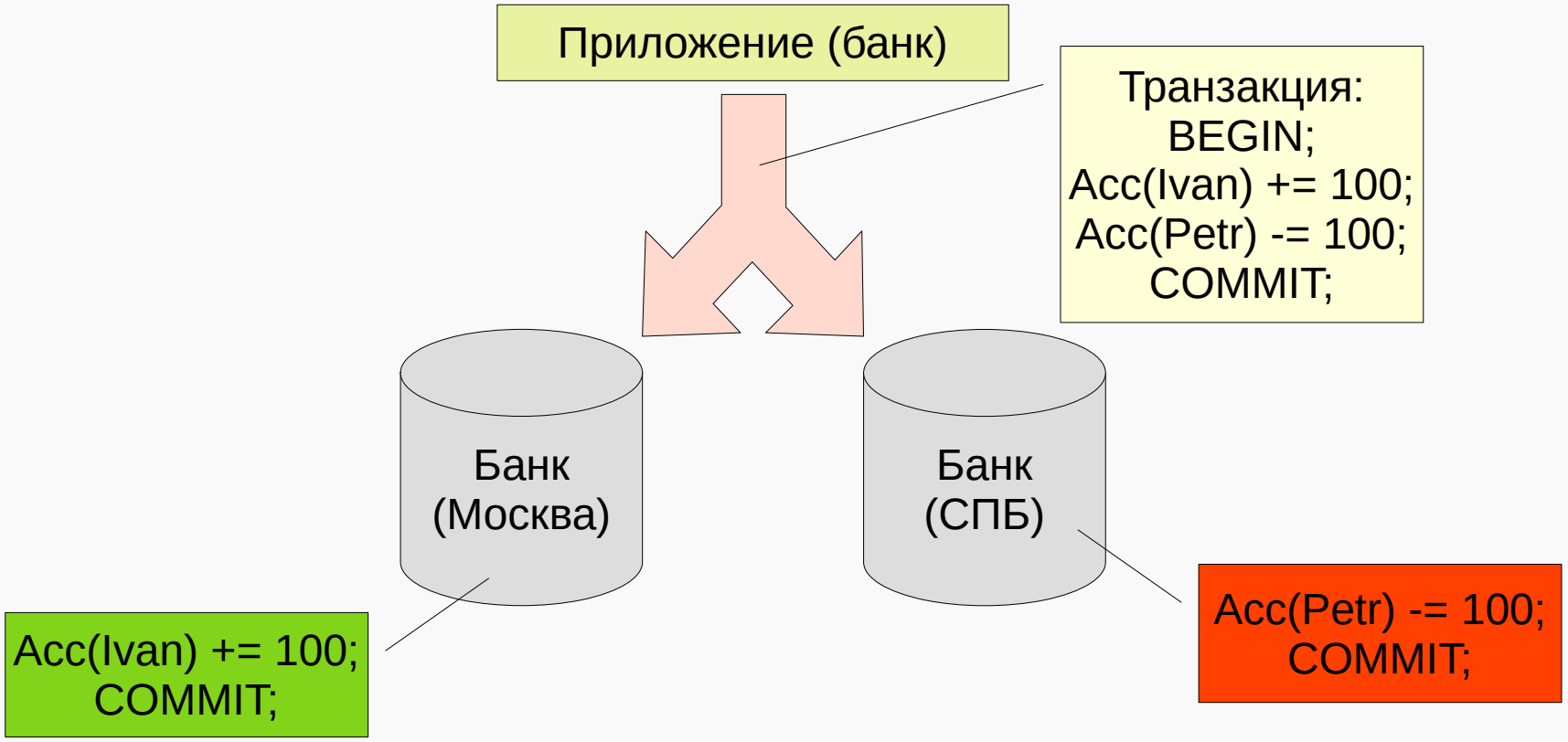
ИТМО ВТ

Пример системы

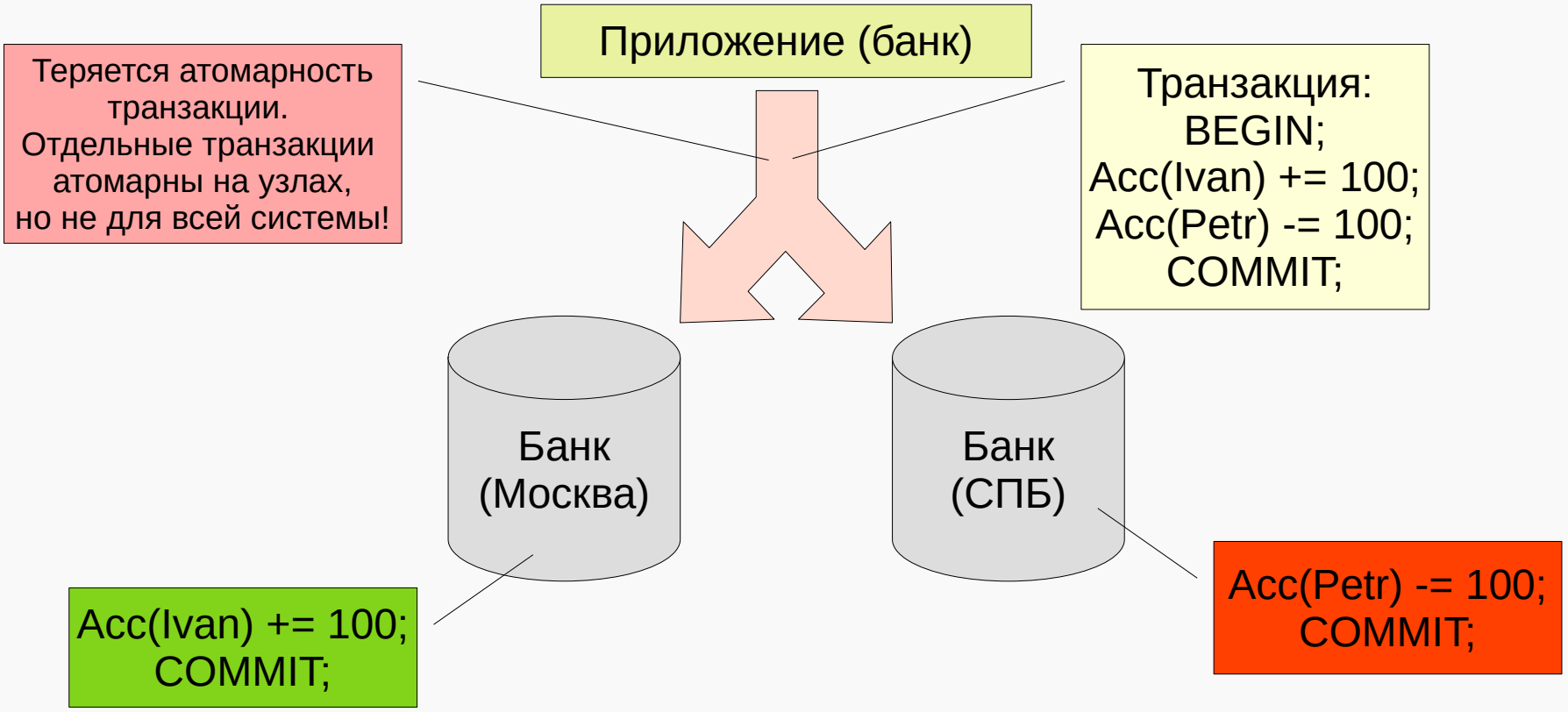


ИТМО ВТ

Пример системы



Пример системы



ИТМО ВТ

Проблема с фиксацией транзакций

- Отдельные узлы — работают, как раньше (атомарность на уровне узлов сохраняется)
- Проблема с системой в целом, так как:
  - на части узлов заданные операции могут не выполниться;
  - часть узлов может оказаться недоступной.
- Нужен определенный протокол для определения возможности фиксации транзакций.

ИТМО ВТ

Двухфазная фиксация

- Двухфазная фиксация (two-phase commit) — протокол для реализации фиксации распределенных транзакций.

Результат транзакции:

- Транзакция успешна (завершена), если зафиксированы все ее компоненты (на всех узлах).
- Если компонент был прерван, **не** выполнился — транзакция не должна быть успешно завершена, аборт всех компонентов.

ИТМО ВТ

Двухфазная фиксация

- Предполагается, что на каждом узле работает свой менеджер ресурсов.
- 2PC нужен в случае, если:
  - на каждом узле свой лог (WAL) для фиксации операций и транзакций;
    - узлы могут быть на одной машине;
  - на узлах могут быть разные данные;
  - если лог разделяем между узлами — 2PC не нужен.

ИТМО ВТ

Двухфазная фиксация

- Узлы делятся на 2 типа: координатор и участники.
- **Координатор** — компонент, который инициирует фиксацию (commit) транзакции и взаимодействует с участниками для определения возможности фиксации распределенной транзакции.
- **Участник** — остальные компоненты, осуществляющие выполнение распределенной транзакции.
  - участник находится в статусе готов, если его изменения зафиксированы.
- Координатор и участники **обмениваются** сообщениями.

Протокол двухфазной фиксации (1)

Фаза 1:

1)Координатор шлет участникам «Запрос на готовность» и ждет их ответа.

2)Через некоторое время участники отвечают координатору:

- «Подготовлен» (prepared), если участник готов зафиксировать транзакцию. Участник переходит в состояние precommit.
- «Нет», если участник не может выполнить свою транзакцию.

Протокол двухфазной фиксации (2)

Фаза 2:

1)Формирование решение координатора:

- Если координатор получает «Подготовлен» от всех участников, принимается решение зафиксировать транзакцию - Commit.
- Если координатор получает «Нет» от одного из участников или не получает от кого-то ответа — решение Abort.

2)Координатор отправляет решение всем участникам.

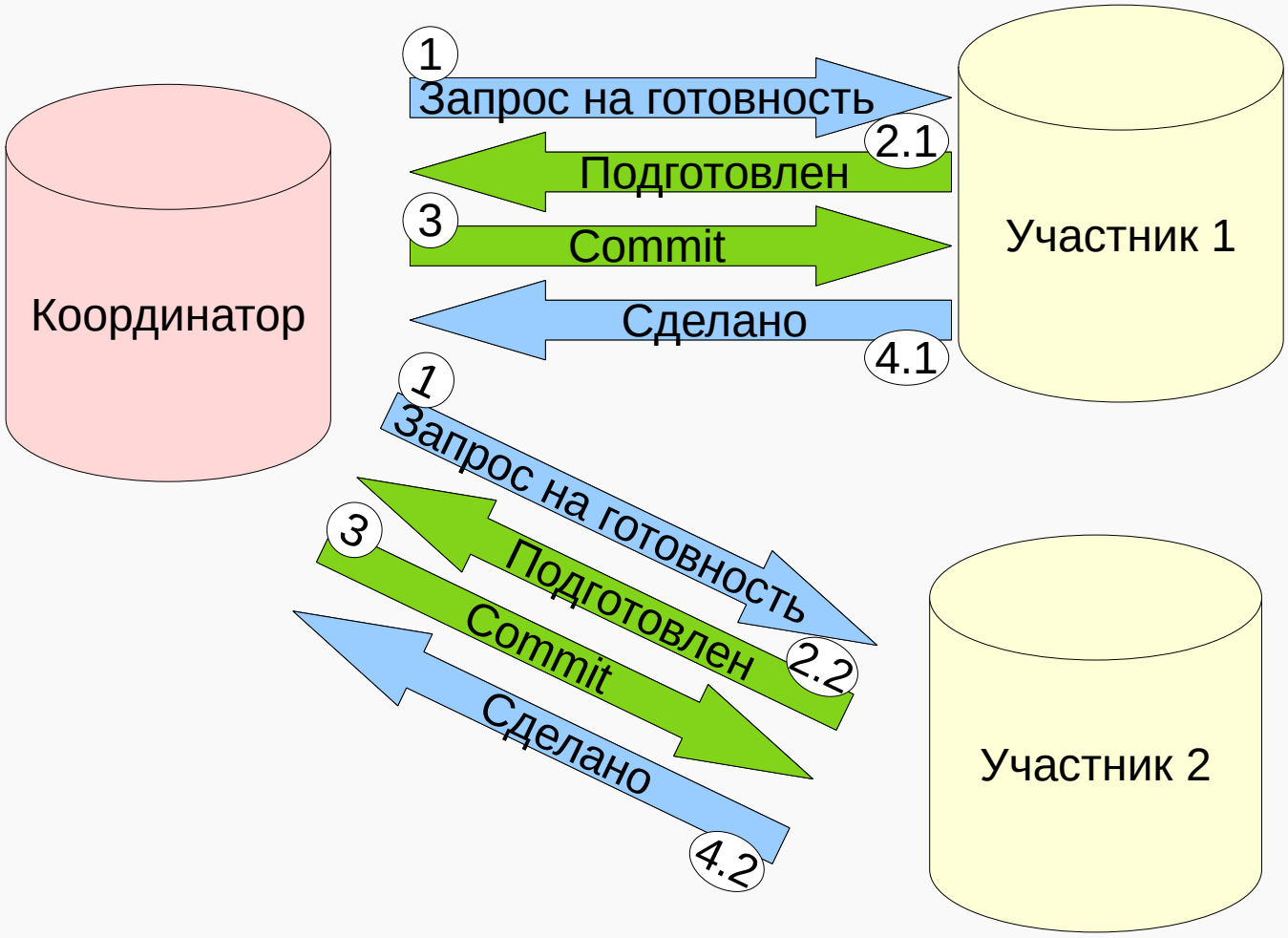
3)Участник получает решение:

- Commit — участник фиксирует транзакцию.
- Abort — участник отменяет транзакцию.

4) После осуществления решения участники отпавляет сообщение о выполнении решения.

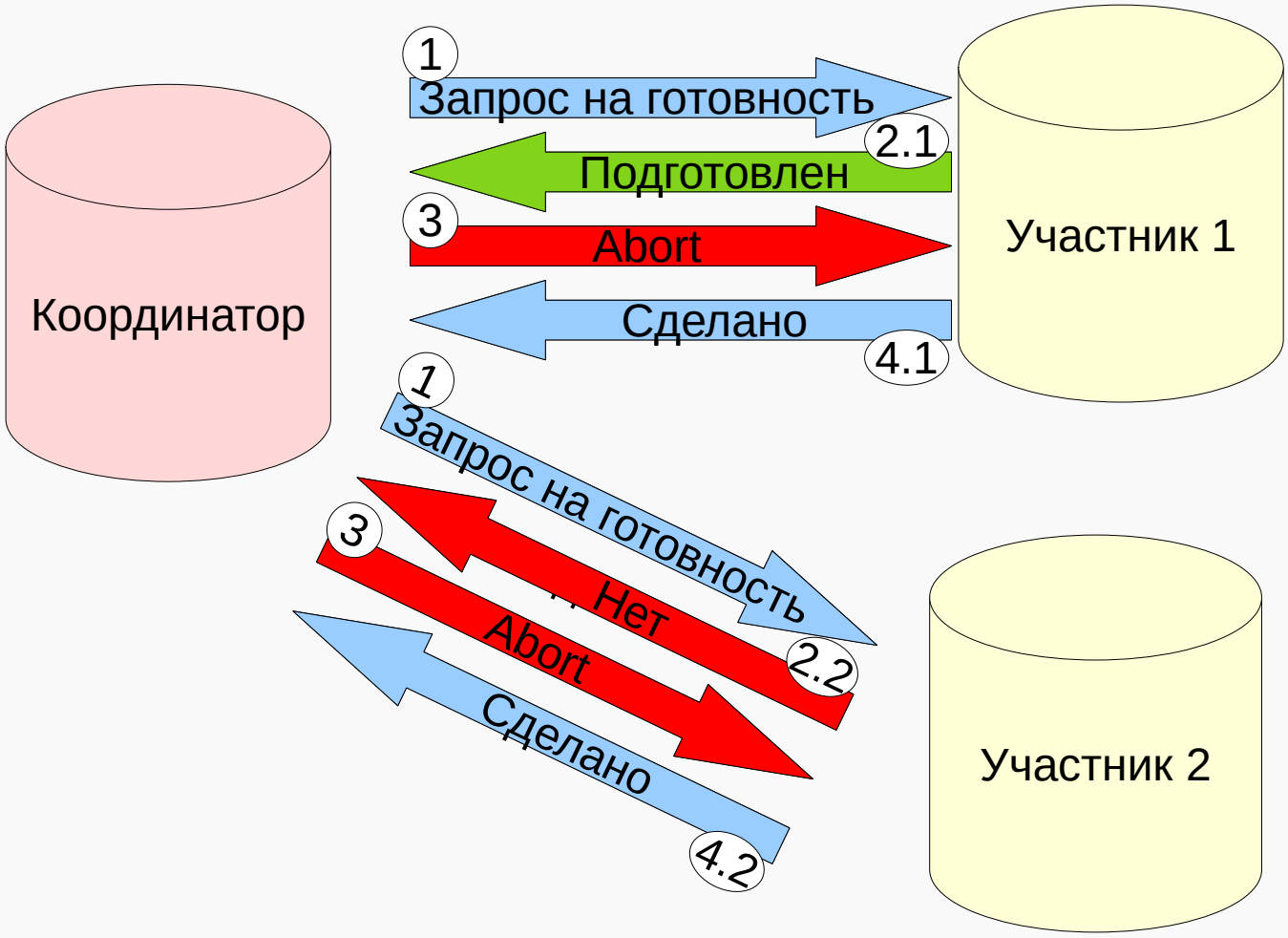
ИТМО ВТ

Протокол двухфазной фиксации: commit

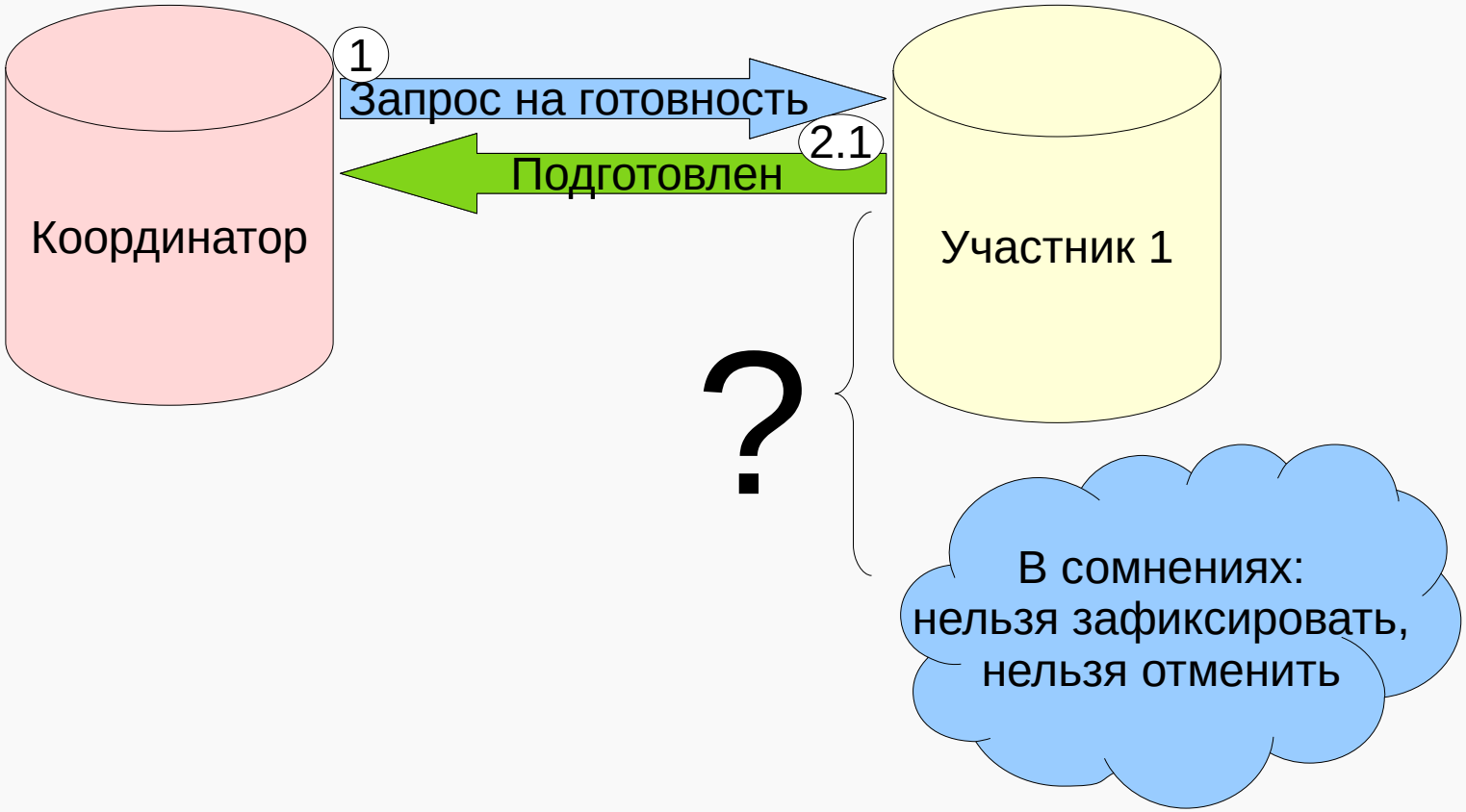


ИТМО ВТ

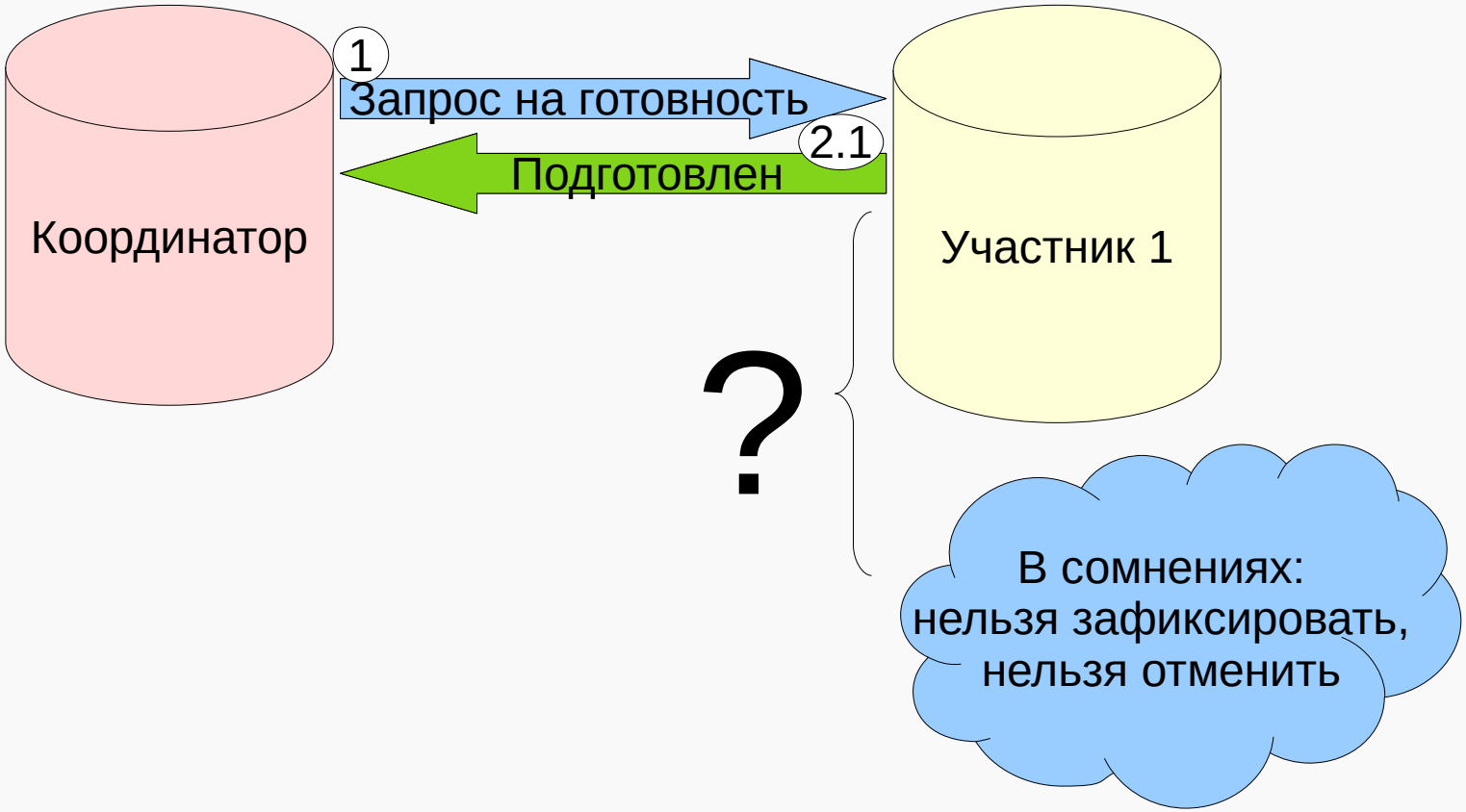
Протокол двухфазной фиксации: abort



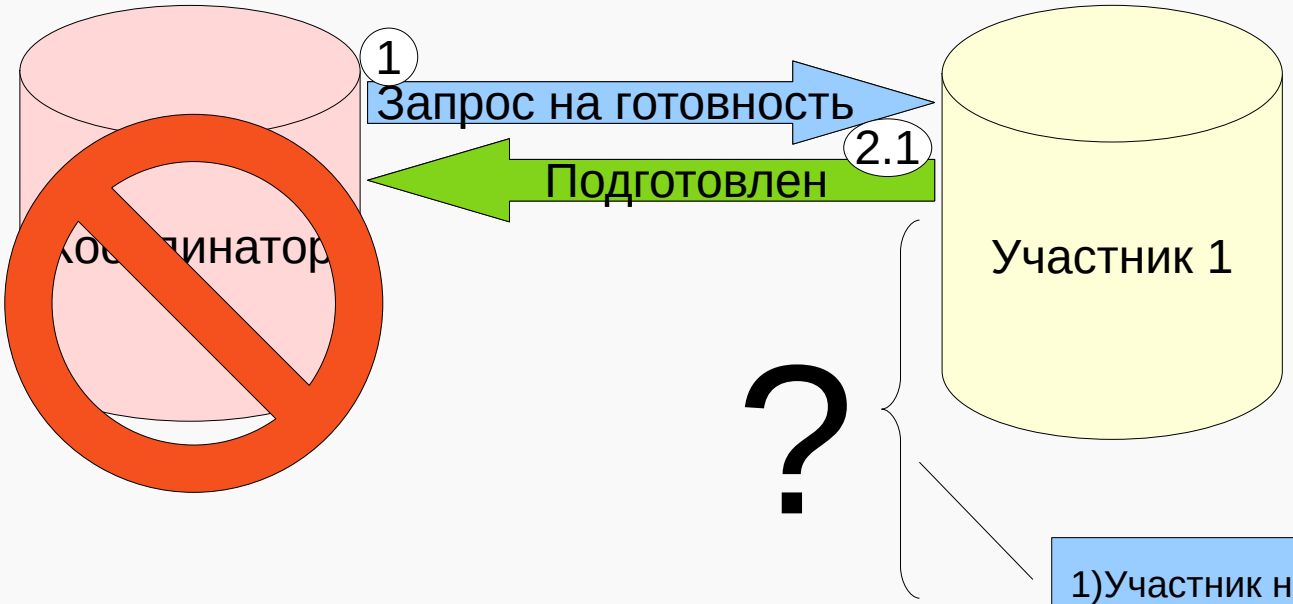
Протокол двухфазной фиксации: «in doubt» (1)



Чем плохо такое состояние? (2)



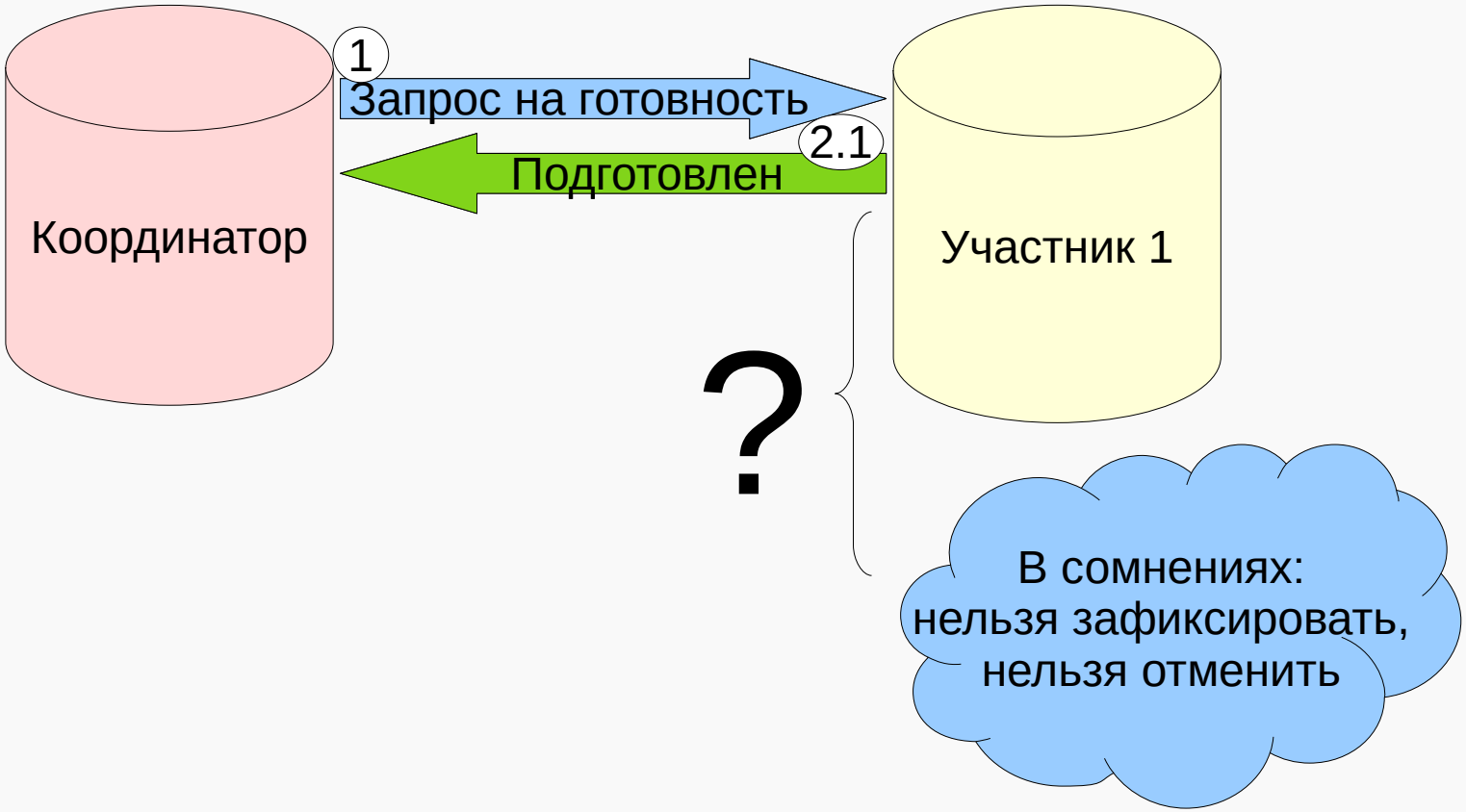
Чем плохо такое состояние? (3)



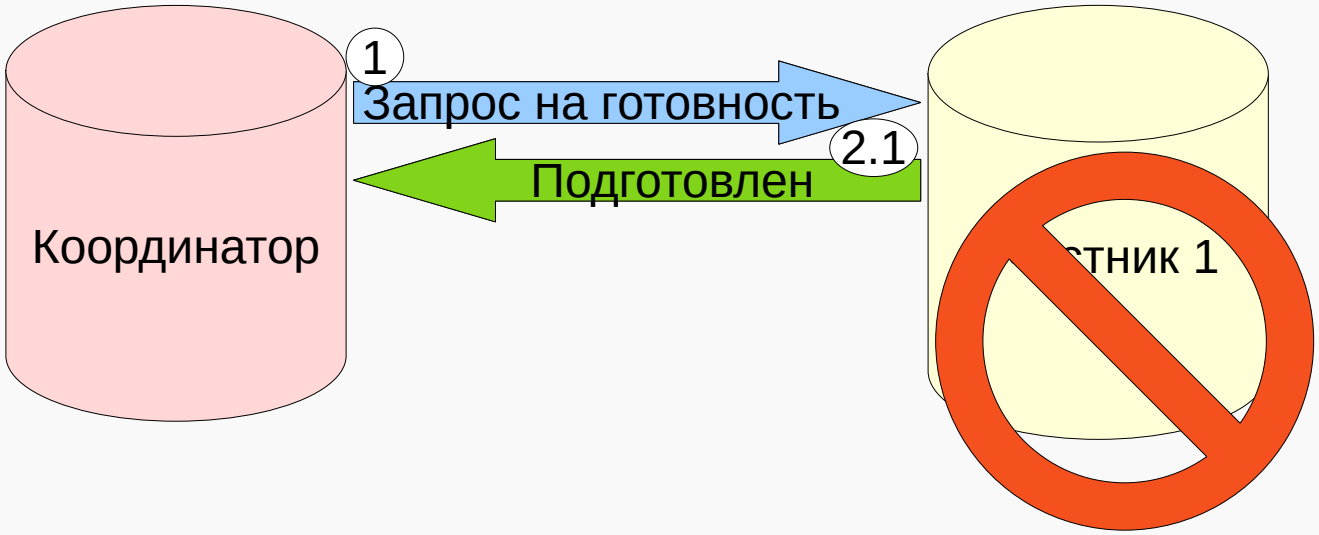
1)Участник не знает состояние координатора.
2)Завершить транзакцию участник не может.
3)Ресурсы, используемые транзакцией, заблокированы.

ИТМО ВТ

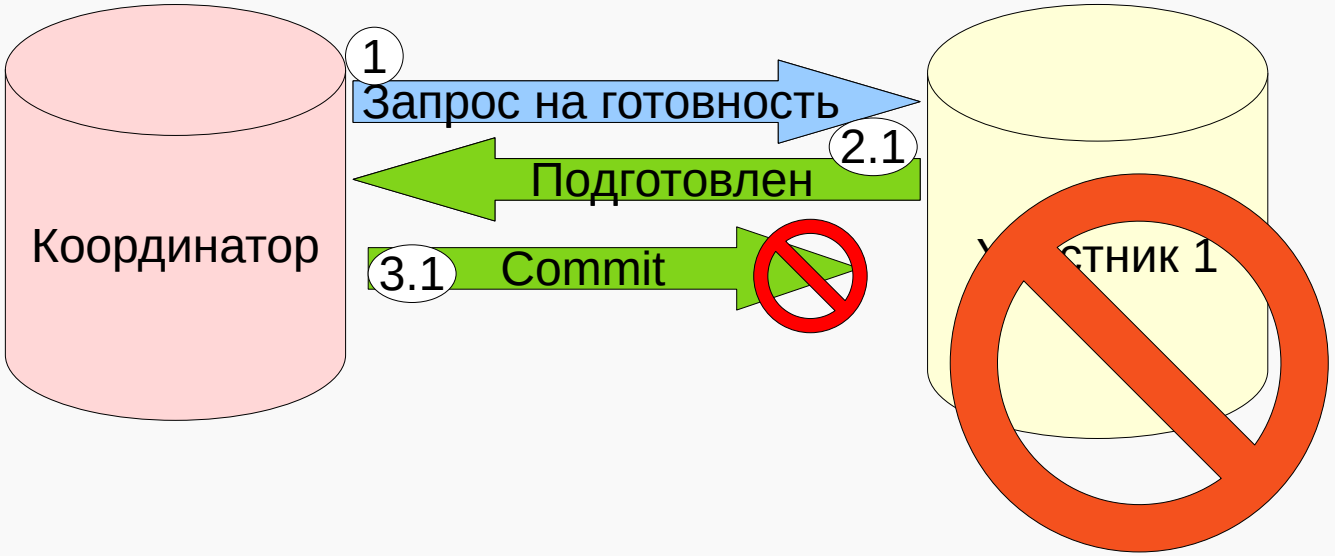
Другой сценарий (1)



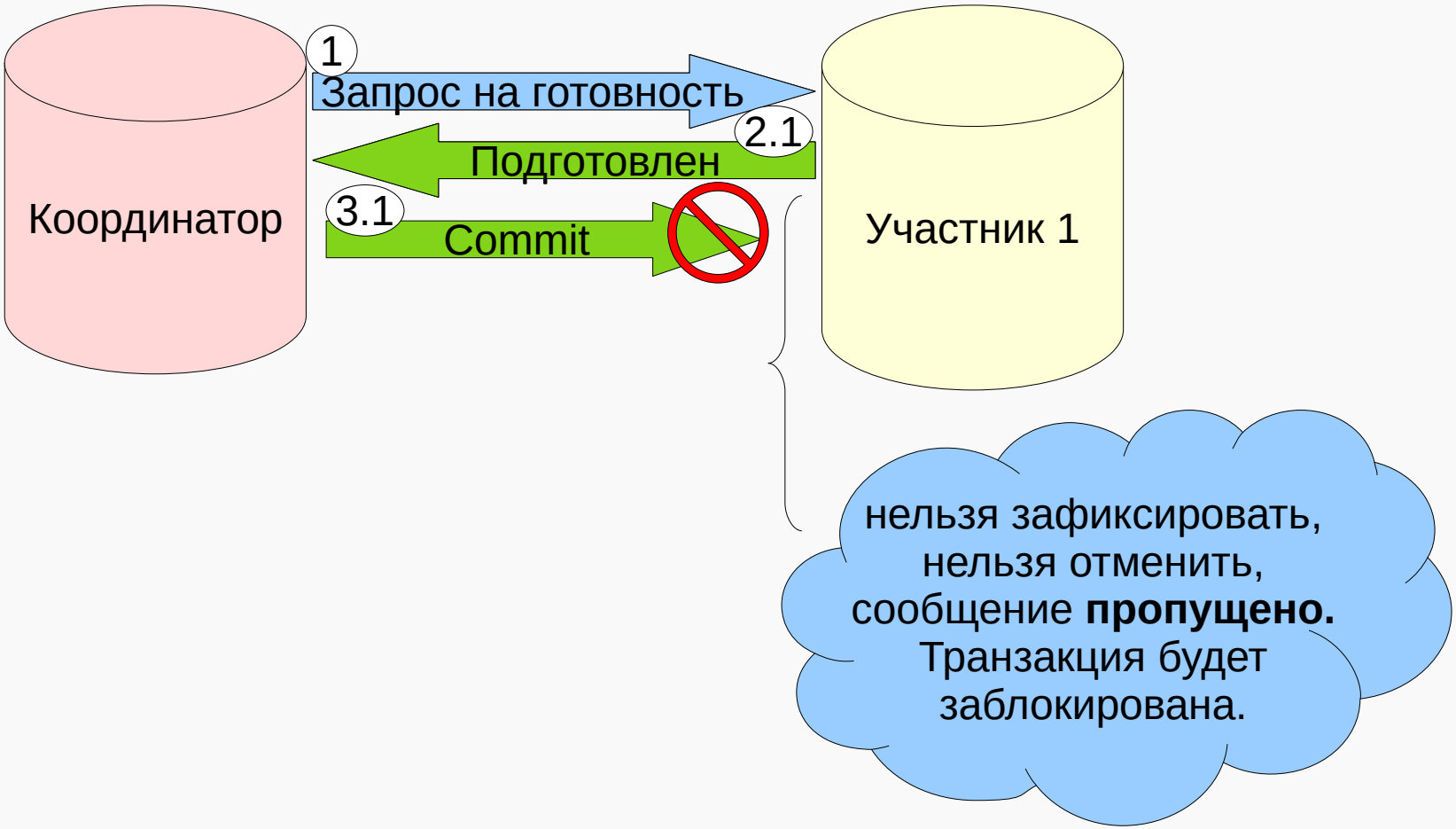
Другой сценарий (2)



Другой сценарий (3)



Другой сценарий (4)



ИТМО ВТ

Обработка отказов

Что нужно для корректной обработки ситуаций, когда происходят сбои участников и координатора?

ИТМО ВТ

Обработка отказов

Что нужно для корректной обработки ситуаций, когда происходят сбои участников и координатора?

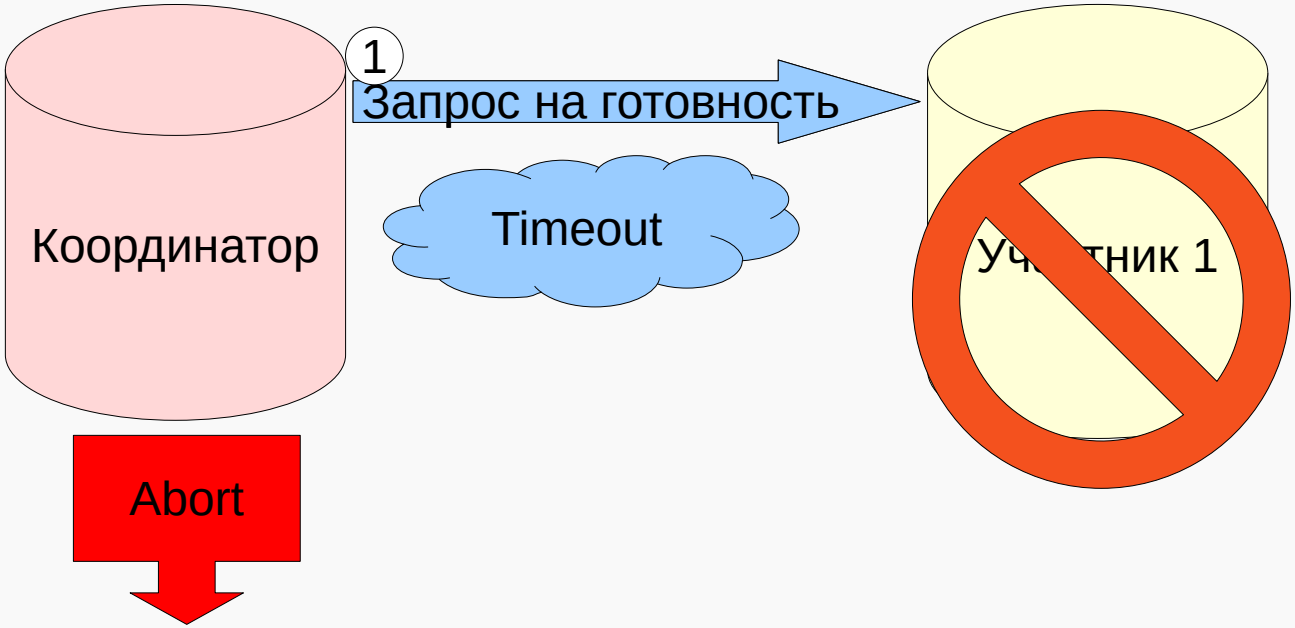
1)Введение Timeout'ов. Ждем некоторый промежуток времени ответа:

- ✗ Ответ не получен: делаем abort или посылаем сообщение-напоминание и устанавливаем новый timeout.

2)Запись в логи об операциях, связанных с фиксацией транзакций.

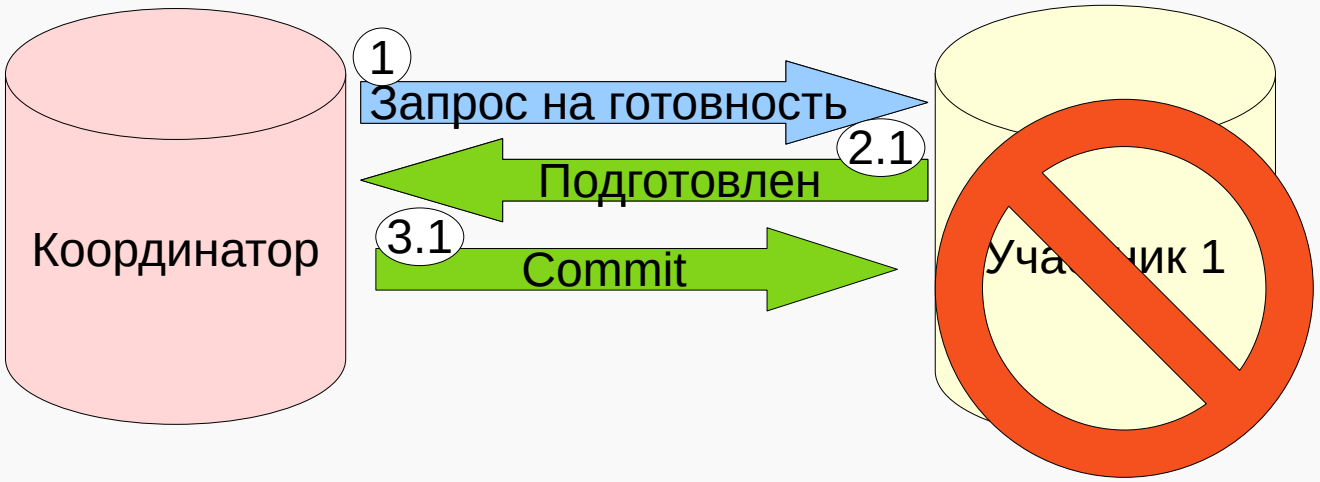
Обработка отказов: координатор (1)

Ожидание сообщений «Подготовлен» или «Нет» от участников. Если сообщение от кого-то не получено — ожидаем timeout. Далее — abort.



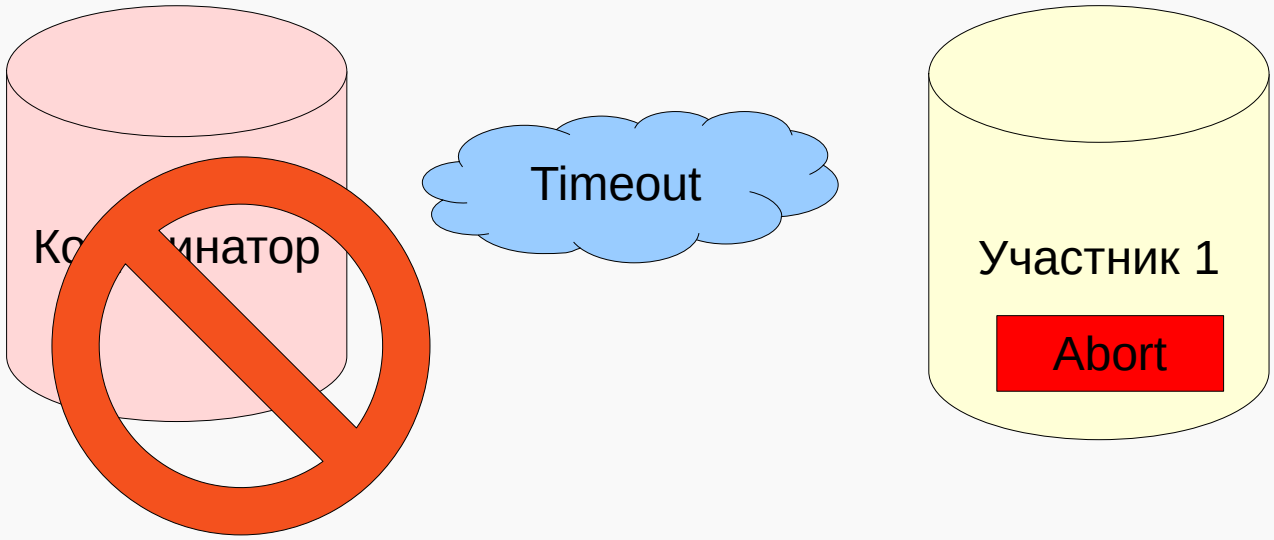
Обработка отказов: координатор (2)

Ожидание сообщений «Готов» от участников. Если сообщение от кого-то не получено — отправить напоминание.



Обработка отказов: участник (1)

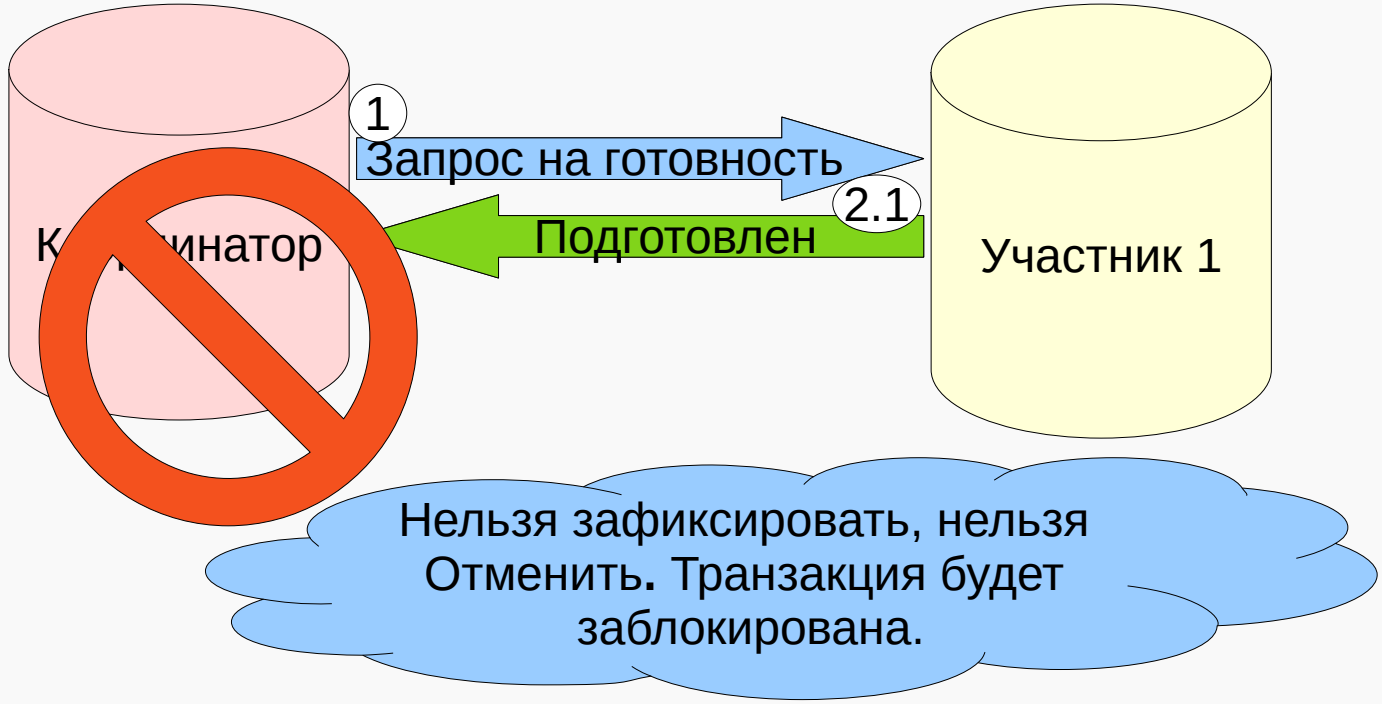
Ожидание сообщения «Запрос на готовность». Если прошел timeout — участник делает аборт транзакции. Если сообщение придет потом, участние ответит «Нет».



Обработка отказов: участник (2)

Блокирован, так как уже подтвердил свою готовность. Ничего не сделать.

Решение: протокол ликвидирования (termination protocol).



ИТМО ВТ

Логирование - координатор

- Перед отсылкой запроса на готовность: **запись о начале 2PC**. Запрос может быть отправлен только тогда, когда запись попадет в журнал.
- Перед отсылкой решения о фиксации: **запись о фиксации**. Решение может быть отправлено только тогда, когда запись попадет в журнал.
- После получения сообщений «Сделано»: **запись о готовности**.

ИТМО ВТ

Логирование - участник

- После получения запроса на готовность: **запись о получении запроса на подготовку**. Запись должна попасть в журнал перед отправкой сообщения о готовности.
- После получения решения о фиксации от координатора: **запись о фиксации или аборте транзакции**. Запись должна попасть в журнал перед отправкой сообщения о готовности.

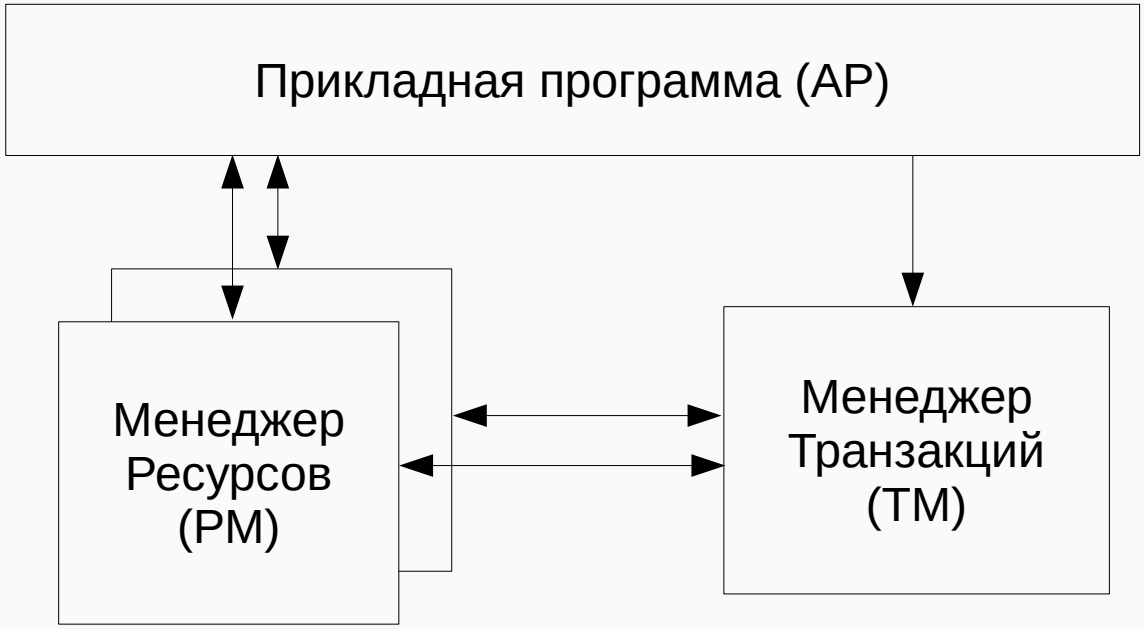
ИТМО ВТ

XA

- XA, X/OpenXA (eXtended Architecture) – спецификация распределённых транзакций.
- Впервые описана в 1992 г., является индустриальным стандартом в реализации менеджеров транзакций.
- Определяет принципы взаимодействия с транзакционными ресурсами в распределённых системах.

ИТМО ВТ

Модель XA



ИТМО ВТ

Элементы модели XA

- Менеджеры ресурсов (RM) – управляют конкретными ресурсами (например, СУБД).
- Менеджер транзакций (TM, он же коодинатор) – координирует работу RM и принимает решение о фиксации или откате транзакции.
- Прикладная программа (AP) – описывает бизнес-логику, управляет составом транзакций и используемыми в ней RM.
- В XA специфицировано только взаимодействие между RM и TM, а за AP отвечает программист.

ИТМО ВТ

Виды транзакций в XA

- Локальные – используется только одно хранилище, роль TM отводится AP.
- Глобальные (распределённые):
  - Несколько хранилищ.
  - AP задаёт границы транзакций.
  - Управлением транзакциями занимается координатор.
- Вложенные:
  - Транзакция запускается внутри существующей.
  - Изменения фиксируются только при фиксации “верхней” транзакции.

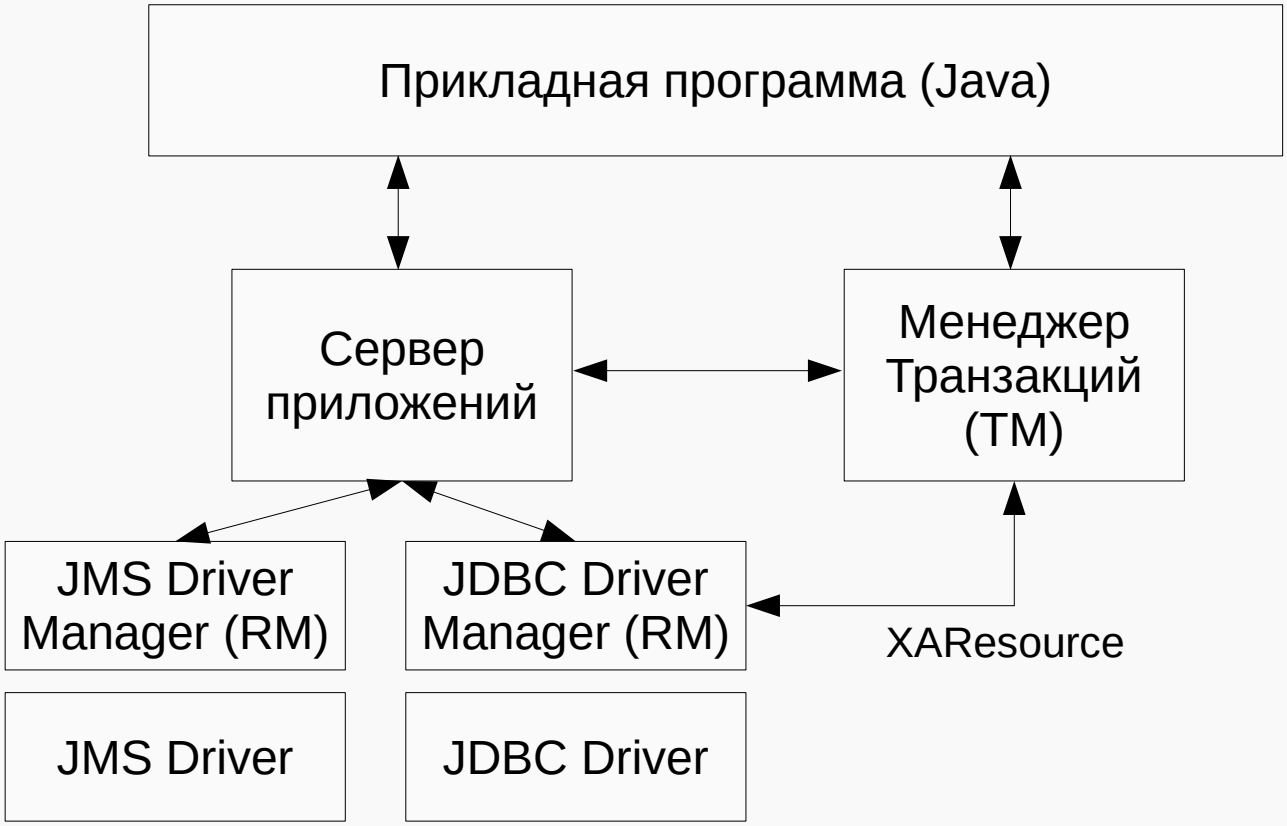
ИТМО ВТ

Java Transaction API

- API для управления транзакциями (в том числе распределенными).
- Специфицирован, входит в состав Java EE.
- Контейнер содержит менеджер транзакций, манипулирующий ресурсами.
- Реализует стандарт X/OpenXA.

ИТМО ВТ

Архитектура JTA



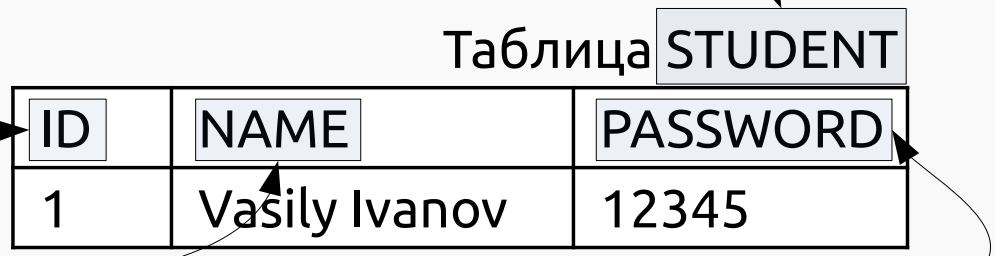
ИТМО ВТ

Особенности JTA

- Поддержка реализуется на уровне сервера приложений (DataSource) и JDBC-драйвера (XAResource и др.).
- На несовместимых с XA JDBC-драйверах распределённые транзакции работать не будут!
- Два высокоуровневых режима управления транзакциями (в EJB) – программный (BMT) и декларативный (CMT).

Student Entity

```
@Entity
public class Student {
    @Id
    int id;

    String name;
    String password;
}
```




```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence persistence_1_0.xsd" version="1.0">
   <persistence-unit name="studentPU" transaction-type="JTA">
      <jta-data-source>studSourceXA</jta-data-source>
   </persistence-unit>
...
</persistence>
```

ИТМО ВТ

JTA: distributed (1)

```
@Stateless
public class StudentEJB {
    @PersistenceContext(unitName="studentPU")
    private EntityManager em1;
        @TransactionAttribute(
            TransactionAttributeType.REQUIRED)
       public void updateStudentName(
                                Integer studId, String name) {
           try {
               Student stud = em1.find(Student.class, studId);
               stud.setName(name);
               em1.persist(stud);
           } catch ( … //catch finally implementation ) { }
    }
}
```

Human Entity

```
@Entity
public class Human {
    @Id
    int id;

    String name;
    String password;
}
```


```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence persistence_1_0.xsd" version="1.0">
    ...
    <persistence-unit name="hrPU" transaction-type="JTA">
        <jta-data-source>hrSourceXA</jta-data-source>
    </persistence-unit>
</persistence>
```

ИТМО ВТ

JTA: distributed (2)

```
@Stateless
public class HREJB {
    @PersistenceContext(unitName="hrPU")
    private EntityManager em2;
        @TransactionAttribute(
            TransactionAttributeType.REQUIRED)
       public void updateHuman(Integer hrId, String name) {
           try {
               Human human = em2.find(Human.class, hrId);
               human.setName(name);
               em2.persist(human);
           } catch ( … //catch finally implementation ) { }
    }
}
```

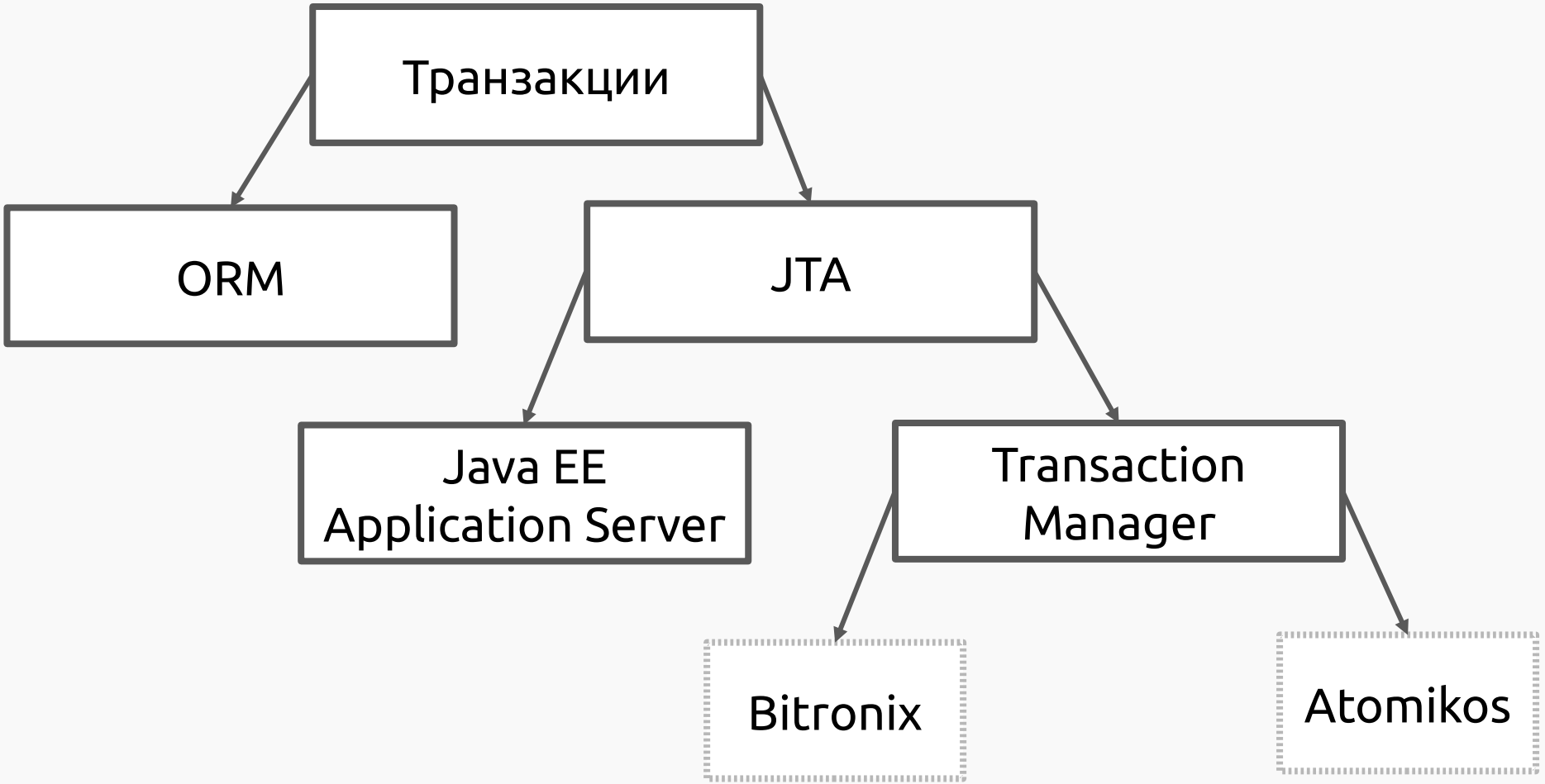
JTA: distributed (3)

```
@Stateless
public class PersonRenamer {
        @EJB
        HREJB hrEJB;

        @EJB
        StudentEJB studentEJB;

        @TransactionAttribute(
            TransactionAttributeType.REQUIRED)
        public void updateHuman(Integer hId, String name) {
           try {
               hrEJB.updateHuman(hId, name);
               studentEJB.updateStudent(hId, name);
           } catch ( … //catch finally implementation ) { }
    }
}
```

4. Транзакции в Spring

ИТМО ВТ

Транзакции в Spring (1)



ИТМО ВТ

Транзакции в Spring (2)

- Как и в Java EE, два режима – программный и декларативный.
- Разработчики Spring рекомендуют использовать декларативный режим.
- Режим “по умолчанию” использует менеджер транзакций ORM-фреймворка.
- Можно использовать транзакции JTA, для этого нужен провайдер – сервер приложений Java EE или отдельный менеджер транзакций.

ИТМО ВТ

Программное управление транзакциями в Spring

Два варианта действий:

- Используем TransactionTemplate, в нём задаём параметры транзакции.
- Используем TransactionManager напрямую.

ИТМО ВТ

Использование TransactionManager

```
public class ServiceImpl implements Service {
    private PlatformTransactionManager transactionManager;
    public void setTransactionManager(PlatformTransactionManager
                                                    transactionManager) {
        this.transactionManager = transactionManager;
        DefaultTransactionDefinition defTr = new DefaultTransactionDefinition();
        def.setName("myTxName");
        def.setPropagationBehavior(TransactionDefinition.PROPAGATION_REQUIRED);
        TransactionStatus status = txManager.getTransaction(defTr);
        try {
            // my business logic
        } catch (Exception ex) {
            txManager.rollback(status);
            throw ex;
        }
        txManager.commit(status);
    }
}
```

ИТМО ВТ

Декларативное управление транзакциями в Spring

- Похоже на JTA (и может использовать его).
- Включается в XML-конфигурации приложения.
- Используются те же менеджеры транзакций, что и в программном режиме.
- Основной элемент – аннотация @Transactional.

ИТМО ВТ

Активация декларативного управления транзакциями

Происходит в конфигурации приложения:

```
@Configuration
@EnableTransactionManagement
public class AppConfig {
   ...
}
```

ИТМО ВТ

Аннотация @Transactional

- Применяется к интерфейсу, классу или методу (как интерфейса, так и класса).
- Аннотация на уровне метода переопределяет “глобальные” настройки класса (или интерфейса).
- Режим по умолчанию – proxy-target-class="false": перехватываются только “внешние” вызовы методов класса.

ИТМО ВТ

Изоляция транзакций

- Задаётся атрибутом isolation аннотации @Transactional.
- Значение по умолчанию – DEFAULT.
- Другие поддерживаемые режимы – READ_COMMITED, READ_UNCOMMITED, REPEATABLE_READ, SERIALIZABLE.

ИТМО ВТ

Взаимодействие с другими транзакциями

- Задаётся атрибутом propagation аннотации @Transactional.
- Значение по умолчанию – REQUIRED.
- Возможные значения те же, что и в JTA – REQUIRED, REQUIRES_NEW, MANDATORY, SUPPORTS, NOT_SUPPORTED.

Пример сервиса

```
@Service
public class ServiceTest {
  @Transactional
  public void myTest1() {
    myTest2();
  }

  @Transactional(propagation = Propagation.REQUIRES_NEW)
  public void myTest2() { /* impl */ }
}
```

ИТМО ВТ

Spring JTA

- Приложение на базе Spring Boot может работать с распределёнными транзакциями JTA.
- Два варианта реализации:
  - Использование сервера приложений Java EE с поддержкой JTA.
  - Использование встроенного менеджера транзакций с поддержкой JTA.

Менеджеры транзакций

- Специальное приложение или библиотека, реализующее управление распределёнными транзакциями.
- Реализует спецификацию JTA.
- Примеры – Bitronix, Atomikos.
- Вместо него Spring Boot может использовать возможности сервера приложений.

ИТМО ВТ

Bitronix Transaction Manager

- Независимая реализация спецификации JTA 1.1.
- Open Source, лицензия Apache v2.
- Не развивается активно с 2016 г., но совместим с современными версиями Spring и ORM.

ИТМО ВТ

Atomikos Transaction Manager

- Похож на Bitronix.
- Open Source, лицензия Apache v2.
- Активно поддерживается.

ИТМО ВТ

Spring Boot + JTA@AppServer

- Достаточно использовать Full-Profile Java EE Application Server.
- Spring Boot попробует самостоятельно выполнить lookup по стандартным путям java:comp/UserTransaction, java:comp/TransactionManager и т.д.
- Spring Boot попробует автоматически сконфигурировать JMS, используя ресурсы java:/JmsXA и java:/XAConnectionFactory.