Особенности разработки высоконагруженного сервера на Java

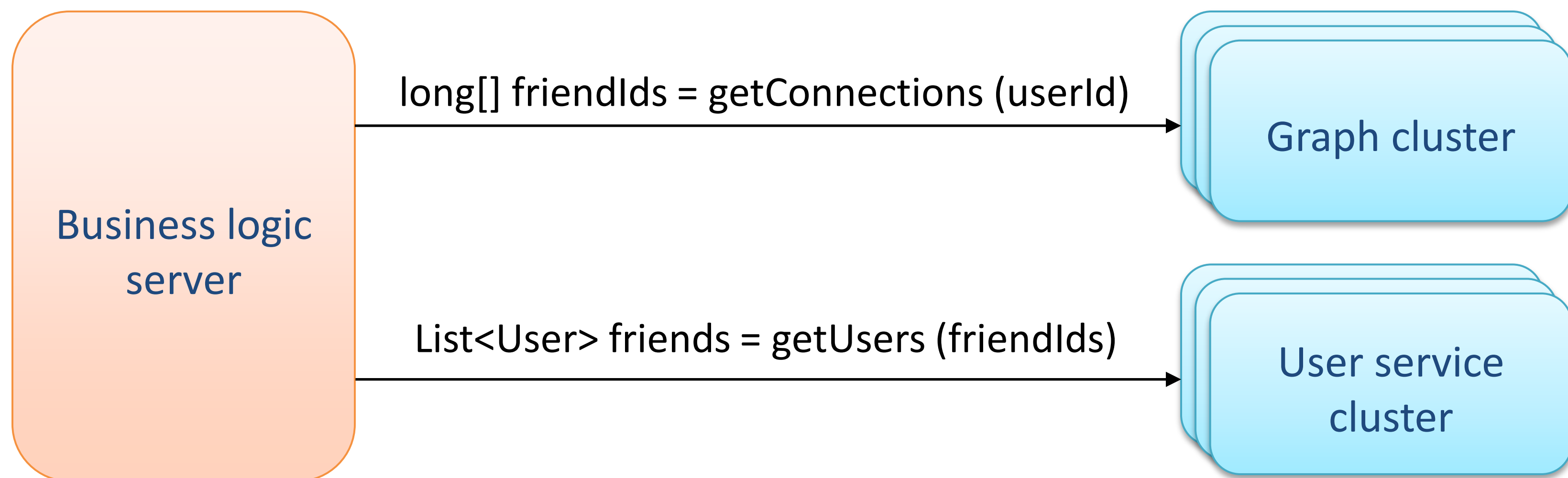
Андрей Паньгин
ведущий разработчик
проекта Одноклассники

Серверы Одноклассников

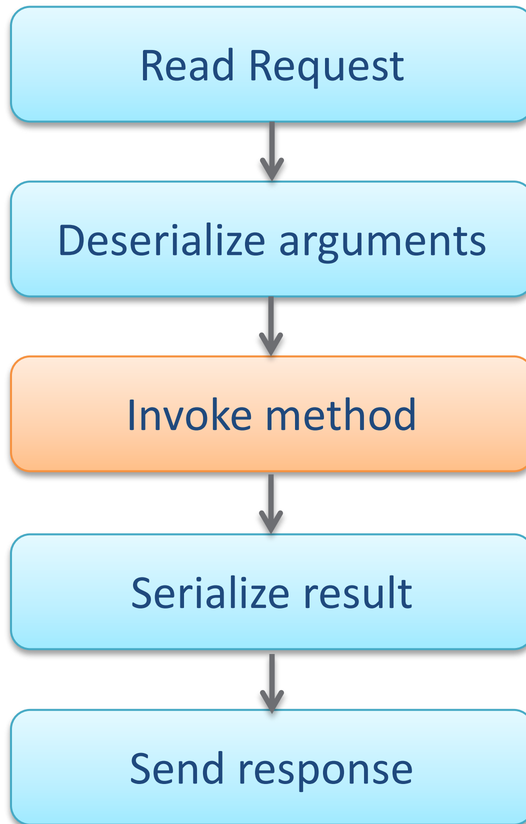
- Всего >3000 серверов
  - Web, Business logic, Download, Storage, Remote Service…
  - Все ПО написано на Java
- Высоконагруженный сервер
  - 20 тыс. одновременных подключений
  - 30 тыс. запросов в секунду
  - Трафик до 1 Gb/s

Remote Service

- Компоненты портала как отдельные сервисы
  - Система сообщений, граф дружб, поиск, лента и др.
- Внутренняя коммуникация между сервисами



RPC сценарий



byte[] request = connection.read(…);

request → Method method, Object[] args

Object result = method.invoke(args);

result → byte[] response

connection.write(response);

java.net I/O (Sockets)

- Поток на каждое соединение
- Максимум 10 тыс. потоков

```
InputStream in = socket.getInputStream();
OutputStream out = socket.getOutputStream();

while (true) {
    int bytesRead = in.read(...);  // blocking call
    if (bytesRead > 0) {
        byte[] response = processRequest(...);
        out.write(response);       // blocking call
    }
}
```

NIO (SocketChannels)

- Selector, неблокирующие read / write
- Direct ByteBuffers

```
while (true) {
    if (selector.select() > 0) {  // blocking call
        Iterator<SelectionKey> iterator =
            selector.selectedKeys().iterator();
        while (iterator.hasNext()) {
            SelectionKey key = iterator.next();
            if (key.isReadable()) {
                doRead(key);
            } else if (key.isWritable()) {
                doWrite(key);
            }
            iterator.remove();
        }
    }
}
```

одноклассники

NIO frameworks

- Apache MINA
  - http://mina.apache.org
- Netty
  - https://netty.io

- Основаны на NIO
- Событийно-управляемая (event-driven) модель

Socket I/O → Netty

- Плюсы
  + Сокращение числа потоков (4000 → 200)
- Минусы
  – Конструирование запроса по частям
  – Влияние на GC
  – Снижение производительности на 20%

Blocking socket selector (BLOS)

- Сочетает преимущества Selector + Blocking I/O
- Возможно в OS, но не поддерживается в Java

Linux	epoll
Solaris	/dev/poll
BSD	kqueue
Windows	I/O completion ports

- Реализуется посредством JNI
  - Простые Java-обертки над системными вызовами

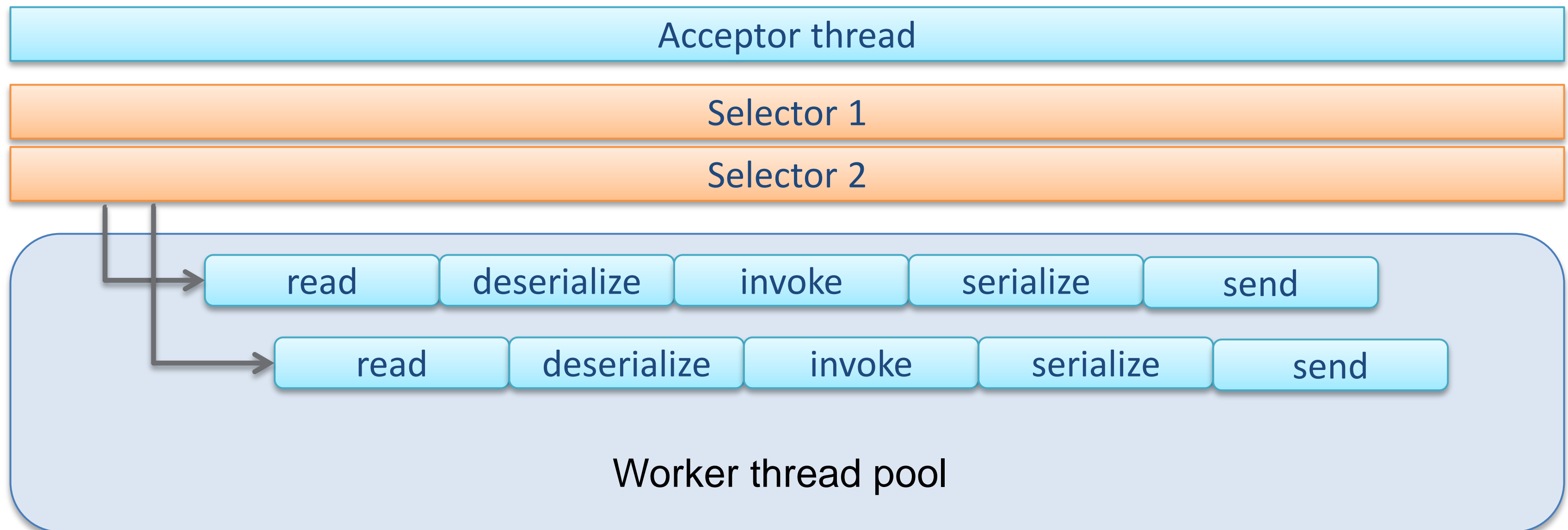
Java Socket → Native

- java.net.Socket
  ↳ java.net.SocketImpl impl
    ↳ java.io.FileDescriptor fd
      ↳ int fd

```
Object getField(Object holder, Class cls, String name) {
    Field f = cls.getDeclaredField(name);
    f.setAccessible(true);
    return f.get(holder);
}
```

Архитектура BLOS сервера

- 1 acceptor thread
- N selector threads (обычно N = кол-во CPU)
- Динамический пул потоков-исполнителей
- Запрос исполняется непрерывно до конца



Проблемы работы с сетью в Java

- java.net.Socket
  - Finalizers => GC impact
    (объекты SocketImpl не собираются до Full GC)
  - Нет поддержки Direct Buffers

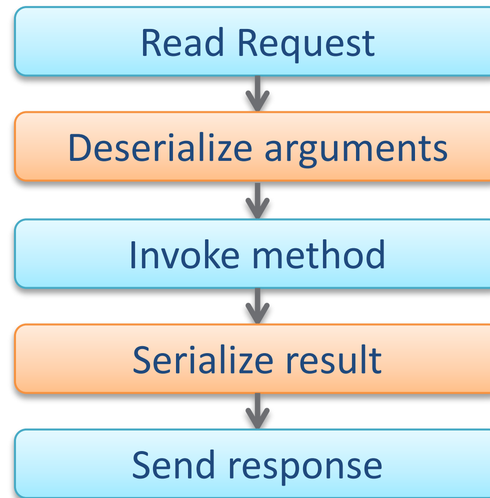
- NIO не спасает
  - Не срабатывают I/O timeouts
  - Возможна утечка нативной памяти

Решение

- И снова JNI
  - Нет finalize(), нет утечки памяти
  - Время GC до 10 раз меньше
  - Необходимость закрывать сокеты вручную
- В Tomcat используется похожий подход
  - APR (Apache Portable Runtime)
    http://tomcat.apache.org/tomcat-7.0-doc/apr.html

Remote Service

- RPC сценарий



- Эффективность сериализации – ключ к производительности RPC
  - Затрачиваемое на сериализацию время
  - Размер передаваемых по сети данных

одноклассники

Сериализация

- Built-in Java Serialization
  - Слишком медленная
  - Большой объем получаемых данных
- JBoss Serialization
  - Не поддерживает разные версии классов
- Ручная сериализация – Externalizable
  - Не вариант (тысячи классов!)

Требования к сериализации

- Быстрая
- Компактная
- Обрабатывает простые изменения внутри класса
  - Добавление поля
  - Удаление поля
  - Изменение порядка полей
  - Изменение типа поля (int → long)

Архитектура сериализации (1/2)

- Типы сериализаторов
  - Встроенные (примитивные типы, обертки, массивы)
  - Collection и Map
  - Сериализация произвольных классов по полям
  - Самосериализуемые классы (readObject / writeObject)

- Каждому сериализуемому классу сопоставляется уникальный 64-битный ID – хеш от имен, типов и порядка полей

Архитектура сериализации (2/2)

- Запись объекта:
  1. Serializer serializer = Repository.getByClass(obj.getClass());
  2. outputStream.writeLong(serializer.uid);
  3. serializer.write(obj, outputStream);

- Чтение объекта:
  1. long uid = inputStream.readLong();
  2. Serializer serializer = Repository.getById(uid);
     - ✓ may throw SerializerNotFoundException
  3. Object obj = serializer.read(inputStream);

Трудности реализации

- Чтение и запись private полей чужих объектов

- Создание экземпляров класса

Трудности реализации

- Чтение и запись private полей чужих объектов
  – Reflection (медленно!)
  – sun.misc.Unsafe

- Создание экземпляров класса
  – sun.misc.Unsafe.allocateInstance()
  – ReflectionFactory.newConstructorForSerialization()
  – Динамическая генерация байткода

Unsafe

```java
import sun.misc.Unsafe;

private static Unsafe getUnsafe() throws Exception {
    Field f = Unsafe.class.getDeclaredField("theUnsafe");
    f.setAccessible(true);
    return (Unsafe) f.get(null);
}

// Instantiate class X without calling X's constructor
// As simple as malloc()
Object x = unsafe.allocateInstance(X.class);

// Read private field f of object x
long offset = unsafe.objectFieldOffset(f);
Object result = unsafe.getObject(x, offset);
```

Генерация байткода

1. Построить массив byte[] с бинарным представлением класса
2. Вызвать ClassLoader.defineClass()
   - ✓ ClassLoader должен быть унаследован

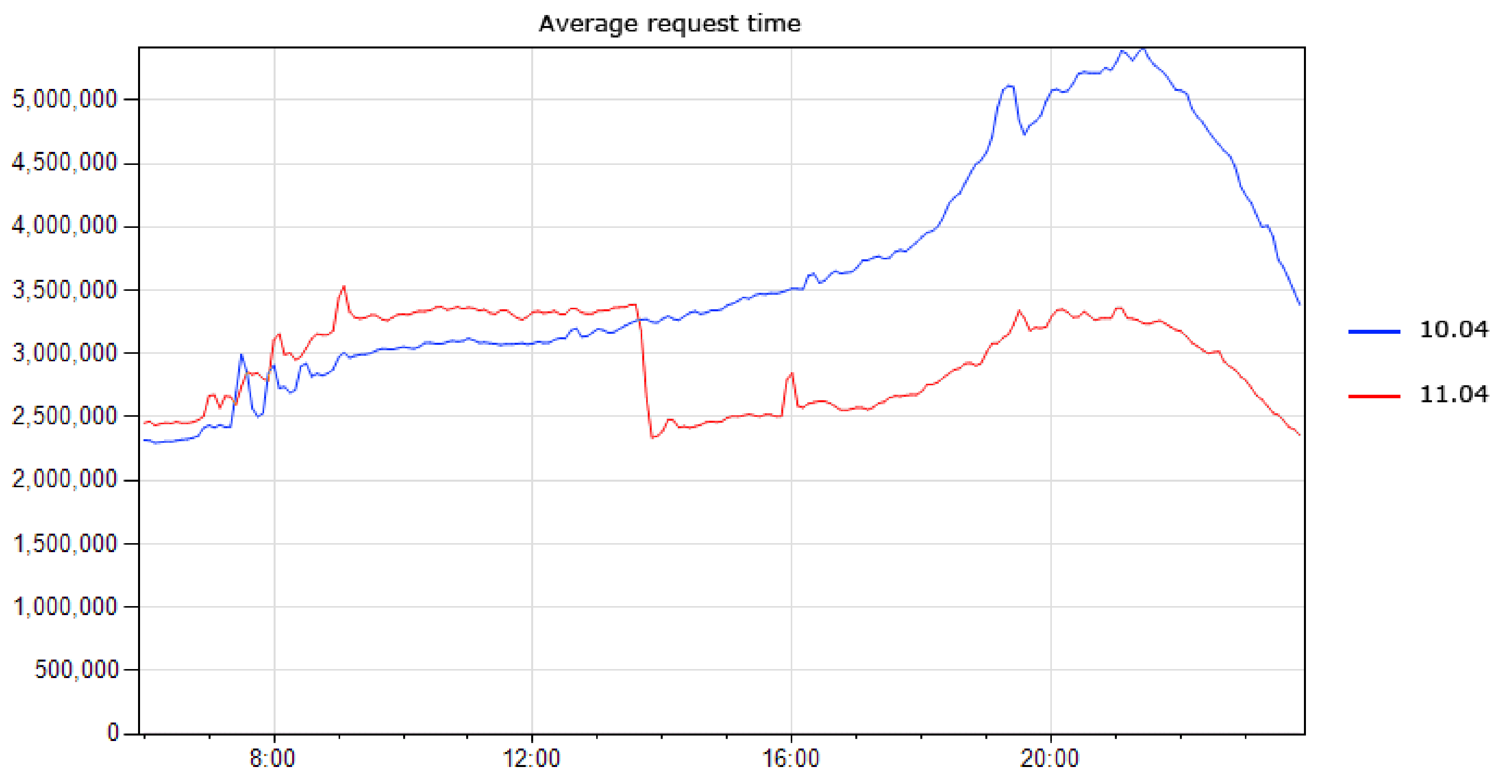
- ASM framework (http://asm.ow2.org)
  - Построить представление класса с помощью org.objectweb.asm.ClassWriter
  - Преобразовать его в массив byte[]: ClassWriter.toByteArray()

А как же private поля?

- JVM при загрузке нового класса верифицирует байткод
- Если класс унаследован от sun.reflect.MagicAccessorImpl, верификатор не будет проверять права доступа для байткодов getfield и putfield
- Reflection внутри использует MagicAccessorImpl

Производительность (1/2)

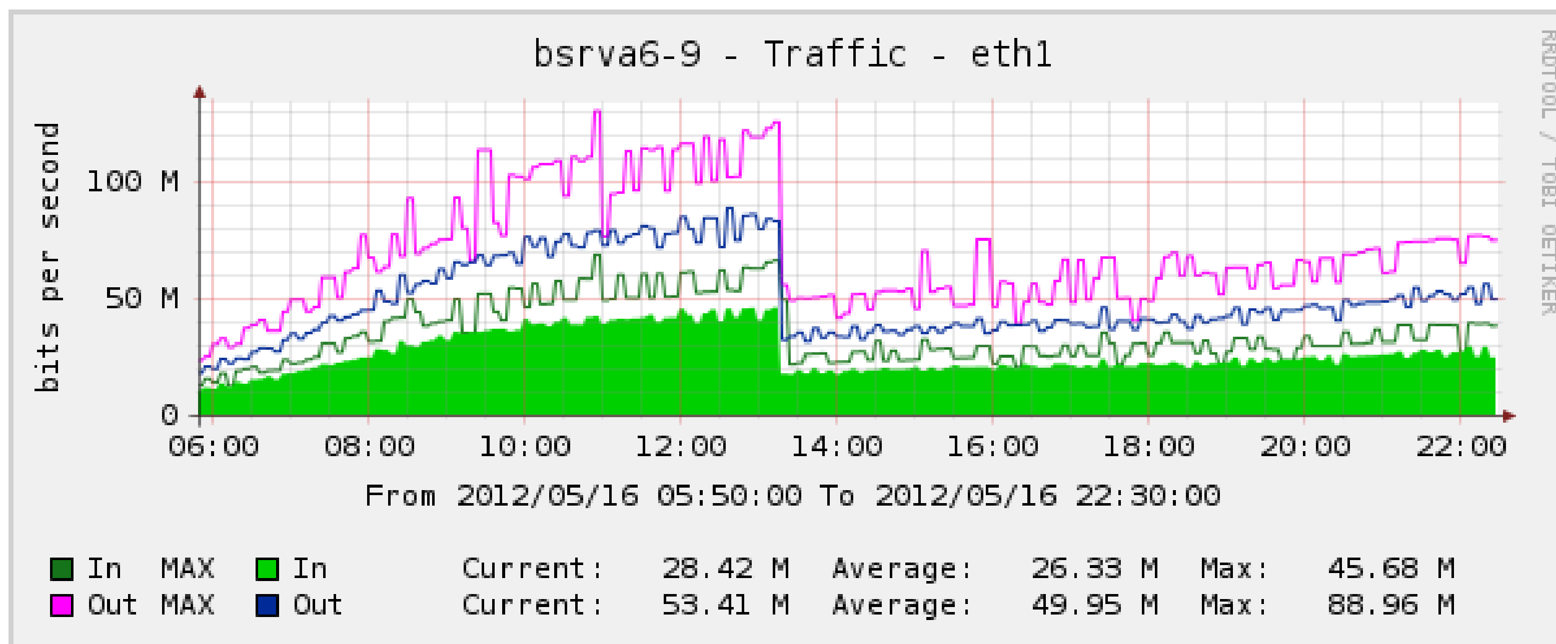
- Количество потоков: 4000 → 200
- Среднее время запроса: -20%



одноклассники

Производительность (2/2)

- GC overhead: -30%
- Сетевой трафик: -50%



Спасибо!

- Примеры
  - https://github.com/odnoklassniki/rmi-samples.git

- Контакты
  - andrey.pangin@odnoklassniki.ru
  - www.odnoklassniki.ru/ap

- В Одноклассниках
  - http://v.ok.ru