# Сетевое программирование

## Лекция 12: Сетевое взаимодействие

**Преподаватель:** [Ваше имя]

**Группа:** 203

**Семестр:** Осенний 2024

# План лекции

1. **Основы сетевого программирования**
2. **TCP и UDP протоколы**
3. **Клиент-сервер архитектура**
4. **Сокеты в Java**
5. **Многопользовательская игра**
6. **Практический пример: Сетевой движок**

# Основы сетевого программирования

## Что такое сетевое программирование?

**Сетевое программирование** — создание приложений, которые могут обмениваться данными по сети.

## Компоненты сети:

- **IP адреса** — идентификация устройств
- **Порты** — идентификация приложений
- **Протоколы** — правила обмена данными
- **Сокеты** — программные интерфейсы для связи

## Применение в играх:

# TCP vs UDP протоколы

## TCP (Transmission Control Protocol):

- **Надежная доставка** данных
- **Порядок** пакетов гарантирован
- **Проверка ошибок** и повторная отправка
- **Подтверждение** получения
- **Медленнее** UDP

## UDP (User Datagram Protocol):

- **Быстрая доставка** данных
- **Порядок** пакетов не гарантирован
- **Без проверки** ошибок

# Выбор протокола для игры

## TCP подходит для:

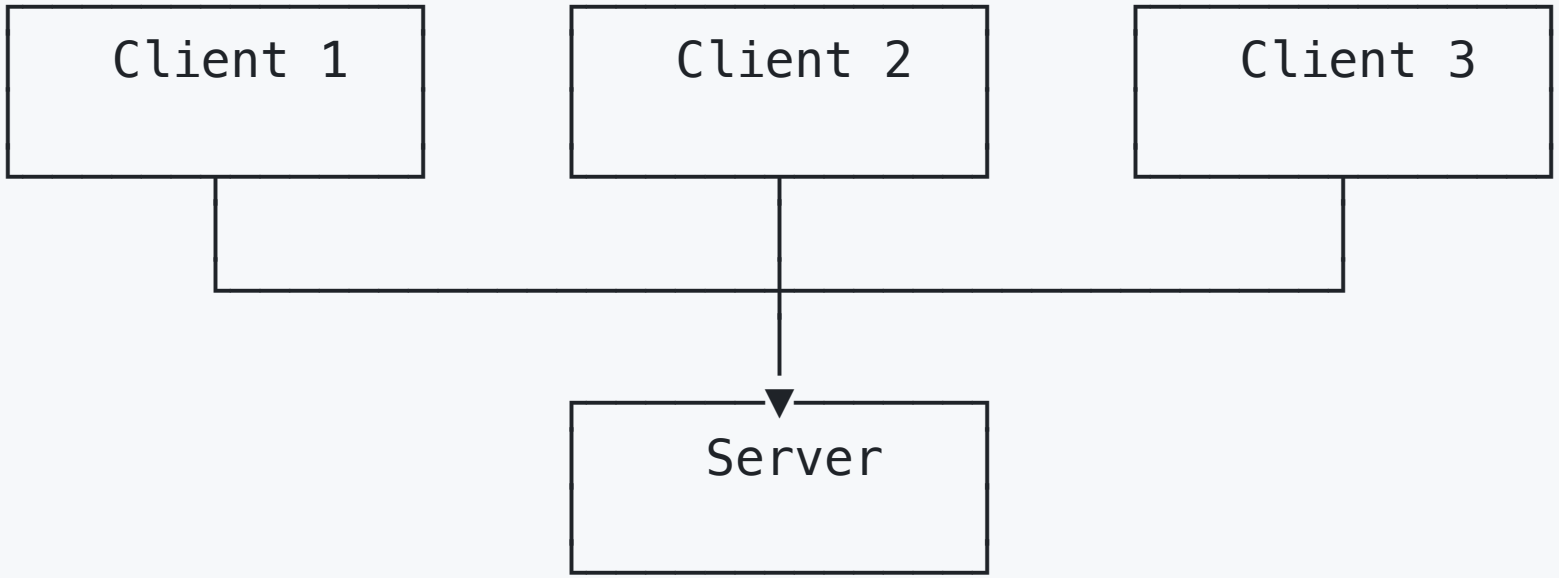
- **Критически важные** данные (логин, покупки)
- **Команды** игры (движение, атака)
- **Состояние** игры (здоровье, ресурсы)
- **Чат** между игроками

## UDP подходит для:

- **Частые обновления** (позиции, анимации)
- **Временные данные** (эффекты, звуки)
- **Потоковые** данные (видео, аудио)
- **Игровые события** (взрывы, частицы)

# Клиент-сервер архитектура

## Структура:




## Роли:

- **Сервер** — управляет игровой логикой, синхронизирует состояние

# Сокеты в Java

## Типы сокетов:

- **Socket** — TCP клиент
- **ServerSocket** — TCP сервер
- **DatagramSocket** — UDP клиент/сервер

## Основные операции:

- **Создание** сокета
- **Подключение** (для клиента)
- **Ожидание подключений** (для сервера)
- **Отправка/получение** данных
- **Закрытие** соединения

# TCP Сервер

```
public class GameServer {
    private ServerSocket serverSocket;
    private List<ClientHandler> clients;
    private GameEngine gameEngine;
    private volatile boolean running;

    public GameServer(int port) {
        this.clients = new ArrayList<>();
        this.gameEngine = new GameEngine();
        this.running = true;

        try {
            this.serverSocket = new ServerSocket(port);
            System.out.println("Сервер запущен на порту " + port);
        } catch (IOException e) {
            System.err.println("Ошибка запуска сервера: " + e.getMessage());
        }
    }

    public void start() {
        // Основной цикл сервера
        while (running) {
            try {
                // Ожидание подключения клиента
                Socket clientSocket = serverSocket.accept();
                System.out.println("Новое подключение: " + clientSocket.getInetAddress());

                // Создание обработчика для клиента
                ClientHandler clientHandler = new ClientHandler(clientSocket, this);
                clients.add(clientHandler);

                // Запуск обработчика в отдельном потоке
                new Thread(clientHandler).start();

            } catch (IOException e) {
                if (running) {
                    System.err.println("Ошибка принятия подключения: " + e.getMessage());
                }
            }
        }
    }

    public void broadcastToAll(String message) {
        clients.removeIf(client -> !client.isConnected());

        for (ClientHandler client : clients) {
            client.sendMessage(message);
        }
    }

    public void broadcastGameState(GameState gameState) {
        String gameStateJson = convertGameStateToJson(gameState);
        broadcastToAll("GAME_STATE:" + gameStateJson);
    }

    public void shutdown() {
        running = false;

        // Отключение всех клиентов
        for (ClientHandler client : clients) {
            client.disconnect();
        }

        // Закрытие сервера
        try {
            if (serverSocket != null && !serverSocket.isClosed()) {
                serverSocket.close();
            }
        } catch (IOException e) {
            System.err.println("Ошибка закрытия сервера: " + e.getMessage());
        }
    }

    private String convertGameStateToJson(GameState gameState) {
        // Конвертация состояния игры в JSON
        ObjectMapper mapper = new ObjectMapper();
        try {
            return mapper.writeValueAsString(gameState);
        } catch (JsonProcessingException e) {
            return "{}";
        }
    }
}
```

# TCP Клиент

```
public class GameClient {
    private Socket socket;
    private BufferedReader in;
    private PrintWriter out;
    private volatile boolean connected;
    private GameController gameController;

    public GameClient(String serverAddress, int port) {
        this.gameController = new GameController();
        connectToServer(serverAddress, port);
    }

    private void connectToServer(String serverAddress, int port) {
        try {
            socket = new Socket(serverAddress, port);
            in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
            out = new PrintWriter(socket.getOutputStream(), true);
            connected = true;

            System.out.println("Подключен к серверу " + serverAddress + ":" + port);

            // Запуск потока для получения сообщений
            startMessageReceiver();

        } catch (IOException e) {
            System.err.println("Ошибка подключения к серверу: " + e.getMessage());
            connected = false;
        }
    }

    private void startMessageReceiver() {
        Thread receiverThread = new Thread(() -> {
            try {
                String message;
                while (connected && (message = in.readLine()) != null) {
                    handleServerMessage(message);
                }
            } catch (IOException e) {
                if (connected) {
                    System.err.println("Ошибка получения сообщения: " + e.getMessage());
                    disconnect();
                }
            }
        });

        receiverThread.setDaemon(true);
        receiverThread.start();
    }

    private void handleServerMessage(String message) {
        if (message.startsWith("GAME_STATE:")) {
            // Обновление состояния игры
            String gameStateJson = message.substring(11);
            GameState gameState = parseGameStateFromJson(gameStateJson);
            gameController.updateGameState(gameState);

        } else if (message.startsWith("CHAT:")) {
            // Сообщение чата
            String chatMessage = message.substring(5);
            gameController.displayChatMessage(chatMessage);

        } else if (message.equals("PLAYER_JOINED")) {
            // Игрок присоединился
            gameController.displayMessage("Новый игрок присоединился к игре");

        } else if (message.equals("PLAYER_LEFT")) {
            // Игрок покинул игру
            gameController.displayMessage("Игрок покинул игру");
        }
    }

    public void sendCommand(GameCommand command) {
        if (connected) {
            String commandJson = convertCommandToJson(command);
            out.println("COMMAND:" + commandJson);
        }
    }

    public void sendChatMessage(String message) {
        if (connected) {
            out.println("CHAT:" + message);
        }
    }

    public void disconnect() {
        connected = false;

        try {
            if (out != null) out.close();
            if (in != null) in.close();
            if (socket != null && !socket.isClosed()) {
                socket.close();
            }
        } catch (IOException e) {
            System.err.println("Ошибка отключения: " + e.getMessage());
        }

        System.out.println("Отключен от сервера");
    }

    private GameState parseGameStateFromJson(String json) {
        ObjectMapper mapper = new ObjectMapper();
        try {
            return mapper.readValue(json, GameState.class);
        } catch (JsonProcessingException e) {
            return new GameState();
        }
    }

    private String convertCommandToJson(GameCommand command) {
        ObjectMapper mapper = new ObjectMapper();
        try {
            return mapper.writeValueAsString(command);
        } catch (JsonProcessingException e) {
            return "{}";
        }
    }

    public boolean isConnected() {
        return connected;
    }
}
```

# Обработчик клиента на сервере

```
public class ClientHandler implements Runnable {
    private Socket clientSocket;
    private BufferedReader in;
    private PrintWriter out;
    private GameServer server;
    private String playerName;
    private volatile boolean connected;

    public ClientHandler(Socket socket, GameServer server) {
        this.clientSocket = socket;
        this.server = server;
        this.connected = true;

        try {
            this.in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
            this.out = new PrintWriter(socket.getOutputStream(), true);
        } catch (IOException e) {
            System.err.println("Ошибка создания потоков: " + e.getMessage());
            connected = false;
        }
    }

    @Override
    public void run() {
        try {
            // Отправка приветственного сообщения
            sendMessage("WELCOME:Добро пожаловать в игру!");

            // Основной цикл обработки сообщений
            String message;
            while (connected && (message = in.readLine()) != null) {
                handleClientMessage(message);
            }

        } catch (IOException e) {
            if (connected) {
                System.err.println("Ошибка чтения от клиента: " + e.getMessage());
            }
        } finally {
            disconnect();
        }
    }

    private void handleClientMessage(String message) {
        if (message.startsWith("LOGIN:")) {
            // Обработка входа игрока
            playerName = message.substring(6);
            server.broadcastToAll("PLAYER_JOINED:" + playerName);
            sendMessage("LOGIN_SUCCESS:" + playerName);

        } else if (message.startsWith("COMMAND:")) {
            // Обработка игровой команды
            String commandJson = message.substring(8);
            GameCommand command = parseCommandFromJson(commandJson);

            if (command != null) {
                // Выполнение команды в игровом движке
                server.getGameEngine().executeCommand(command);

                // Отправка обновленного состояния всем игрокам
                GameState updatedState = server.getGameEngine().getGameState();
                server.broadcastGameState(updatedState);
            }

        } else if (message.startsWith("CHAT:")) {
            // Обработка сообщения чата
            String chatMessage = message.substring(5);
            server.broadcastToAll("CHAT:" + playerName + ": " + chatMessage);

        } else if (message.equals("DISCONNECT")) {
            // Отключение игрока
            disconnect();
        }
    }

    public void sendMessage(String message) {
        if (connected) {
            out.println(message);
        }
    }

    public void disconnect() {
        connected = false;

        try {
            if (out != null) out.close();
            if (in != null) in.close();
            if (clientSocket != null && !clientSocket.isClosed()) {
                clientSocket.close();
            }
        } catch (IOException e) {
            System.err.println("Ошибка отключения клиента: " + e.getMessage());
        }

        // Уведомление других игроков
        if (playerName != null) {
            server.broadcastToAll("PLAYER_LEFT:" + playerName);
        }

        // Удаление из списка клиентов
        server.removeClient(this);

        System.out.println("Клиент отключен: " + playerName);
    }

    private GameCommand parseCommandFromJson(String json) {
        ObjectMapper mapper = new ObjectMapper();
        try {
            return mapper.readValue(json, GameCommand.class);
        } catch (JsonProcessingException e) {
            return null;
        }
    }

    public boolean isConnected() {
        return connected;
    }

    public String getPlayerName() {
        return playerName;
    }
}
```

# UDP для игровых событий

```
public class UDPGameServer {
    private DatagramSocket socket;
    private GameEngine gameEngine;
    private volatile boolean running;
    private Map<InetAddress, Integer> clients;

    public UDPGameServer(int port) {
        this.clients = new ConcurrentHashMap<>();
        this.gameEngine = new GameEngine();

        try {
            this.socket = new DatagramSocket(port);
            System.out.println("UDP сервер запущен на порту " + port);
        } catch (SocketException e) {
            System.err.println("Ошибка запуска UDP сервера: " + e.getMessage());
        }
    }

    public void start() {
        running = true;

        while (running) {
            try {
                // Буфер для получения данных
                byte[] buffer = new byte[1024];
                DatagramPacket packet = new DatagramPacket(buffer, buffer.length);

                // Ожидание пакета
                socket.receive(packet);

                // Обработка пакета в отдельном потоке
                new Thread(() -> handlePacket(packet)).start();

            } catch (IOException e) {
                if (running) {
                    System.err.println("Ошибка получения UDP пакета: " + e.getMessage());
                }
            }
        }
    }

    private void handlePacket(DatagramPacket packet) {
        InetAddress clientAddress = packet.getAddress();
        int clientPort = packet.getPort();

        // Регистрация клиента
        clients.put(clientAddress, clientPort);

        // Парсинг сообщения
        String message = new String(packet.getData(), 0, packet.getLength());

        if (message.startsWith("POSITION:")) {
            // Обновление позиции игрока
            handlePositionUpdate(message, clientAddress, clientPort);

        } else if (message.startsWith("EVENT:")) {
            // Обработка игрового события
            handleGameEvent(message, clientAddress, clientPort);
        }
    }

    private void handlePositionUpdate(String message, InetAddress clientAddress, int clientPort) {
        // Парсинг позиции
        String[] parts = message.split(":");
        if (parts.length == 3) {
            try {
                int x = Integer.parseInt(parts[1]);
                int y = Integer.parseInt(parts[2]);

                // Обновление позиции в игровом движке
                gameEngine.updatePlayerPosition(clientAddress, x, y);

                // Отправка обновленной позиции всем клиентам
                broadcastPositionUpdate(clientAddress, x, y);

            } catch (NumberFormatException e) {
                System.err.println("Неверный формат позиции: " + message);
            }
        }
    }

    private void handleGameEvent(String message, InetAddress clientAddress, int clientPort) {
        // Парсинг события
        String[] parts = message.split(":");
        if (parts.length >= 2) {
            String eventType = parts[1];

            switch (eventType) {
                case "ATTACK":
                    handleAttackEvent(message, clientAddress, clientPort);
                    break;
                case "BUILD":
                    handleBuildEvent(message, clientAddress, clientPort);
                    break;
                case "RESOURCE":
                    handleResourceEvent(message, clientAddress, clientPort);
                    break;
            }
        }
    }

    private void broadcastPositionUpdate(InetAddress sourceAddress, int x, int y) {
        String updateMessage = "POSITION_UPDATE:" + sourceAddress.getHostAddress() + ":" + x + ":" + y;
        byte[] messageBytes = updateMessage.getBytes();

        // Отправка всем клиентам кроме источника
        for (Map.Entry<InetAddress, Integer> client : clients.entrySet()) {
            if (!client.getKey().equals(sourceAddress)) {
                DatagramPacket packet = new DatagramPacket(
                    messageBytes, messageBytes.length,
                    client.getKey(), client.getValue()
                );

                try {
                    socket.send(packet);
                } catch (IOException e) {
                    System.err.println("Ошибка отправки UDP пакета: " + e.getMessage());
                }
            }
        }
    }

    public void shutdown() {
        running = false;

        if (socket != null && !socket.isClosed()) {
            socket.close();
        }
    }
}
```

# Практический пример: Сетевой движок игры

```
public class NetworkGameEngine {
    private GameServer tcpServer;
    private UDPGameServer udpServer;
    private GameEngine gameEngine;
    private Map<String, Player> players;
    private volatile boolean running;

    public NetworkGameEngine(int tcpPort, int udpPort) {
        this.players = new ConcurrentHashMap<>();
        this.gameEngine = new GameEngine();
        this.running = true;

        // Запуск TCP сервера для команд и чата
        this.tcpServer = new GameServer(tcpPort);

        // Запуск UDP сервера для позиций и событий
        this.udpServer = new UDPGameServer(udpPort);
    }

    public void start() {
        // Запуск TCP сервера в отдельном потоке
        Thread tcpThread = new Thread(() -> {
            tcpServer.start();
        });
        tcpThread.setDaemon(true);
        tcpThread.start();

        // Запуск UDP сервера в отдельном потоке
        Thread udpThread = new Thread(() -> {
            udpServer.start();
        });
        udpThread.setDaemon(true);
        udpThread.start();

        // Основной игровой цикл
        startGameLoop();
    }

    private void startGameLoop() {
        Thread gameLoopThread = new Thread(() -> {
            while (running) {
                try {
                    // Обновление игровой логики
                    gameEngine.update();

                    // Синхронизация состояния с клиентами
                    if (gameEngine.hasStateChanged()) {
                        GameState currentState = gameEngine.getGameState();
                        tcpServer.broadcastGameState(currentState);
                    }

                    // Пауза между обновлениями
                    Thread.sleep(50); // 20 FPS для сервера

                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                    break;
                }
            }
        });

        gameLoopThread.setDaemon(true);
        gameLoopThread.start();
    }

    public void addPlayer(String playerName, InetAddress address) {
        Player player = new Player(playerName, address);
        players.put(playerName, player);

        // Добавление игрока в игровой движок
        gameEngine.addPlayer(player);

        System.out.println("Игрок добавлен: " + playerName);
    }

    public void removePlayer(String playerName) {
        Player player = players.remove(playerName);
        if (player != null) {
            // Удаление игрока из игрового движка
            gameEngine.removePlayer(player);
            System.out.println("Игрок удален: " + playerName);
        }
    }

    public void executeCommand(String playerName, GameCommand command) {
        Player player = players.get(playerName);
        if (player != null) {
            // Выполнение команды в игровом движке
            gameEngine.executeCommand(player, command);
        }
    }

    public void shutdown() {
        running = false;

        if (tcpServer != null) {
            tcpServer.shutdown();
        }

        if (udpServer != null) {
            udpServer.shutdown();
        }

        System.out.println("Сетевой движок остановлен");
    }

    public GameEngine getGameEngine() {
        return gameEngine;
    }

    public Map<String, Player> getPlayers() {
        return new HashMap<>(players);
    }
}

// Главный класс сервера
public class GameServerMain {
    public static void main(String[] args) {
        int tcpPort = 8888;
        int udpPort = 8081;

        if (args.length >= 2) {
            tcpPort = Integer.parseInt(args[0]);
            udpPort = Integer.parseInt(args[1]);
        }

        NetworkGameEngine engine = new NetworkGameEngine(tcpPort, udpPort);

        // Обработка сигнала завершения
        Runtime.getRuntime().addShutdownHook(new Thread(() -> {
            System.out.println("Получен сигнал завершения...");
            engine.shutdown();
        }));

        try {
            engine.start();
            System.out.println("Сервер запущен. Нажмите Ctrl+C для остановки.");

            // Ожидание завершения
            while (true) {
                Thread.sleep(1000);
            }

        } catch (InterruptedException e) {
            System.out.println("Сервер прерван");
        } finally {
            engine.shutdown();
        }
    }
}
```

# Лучшие практики сетевого программирования

## ✅ Что делать:

- **Использовать TCP** для критически важных данных
- **Использовать UDP** для частых обновлений
- **Обрабатывать ошибки** сети корректно
- **Валидировать** входящие данные
- **Логировать** сетевые события

## ❌ Чего избегать:

- **Блокировать UI поток** сетевыми операциями
- **Игнорировать** таймауты соединений
- **Отправлять** большие объемы данных по UDP

# Домашнее задание

## Задача 1:

Реализовать простой TCP сервер для игры

## Задача 2:

Создать UDP клиент для отправки позиций

## Задача 3:

Реализовать чат между игроками

# Что дальше?

## На следующей лекции:

- **Тестирование**
- **JUnit 5**
- **Mock объекты**
- **Тестирование архитектуры**

## Подготовка:

- Изучить главу 23-24 из учебника
- Выполнить домашнее задание
- Подготовить вопросы по текущей теме

# Вопросы?

## Контакты:

- **Email:** [ваш.email@university.edu]
- **Telegram:** [@username]
- **Офис:** [номер кабинета]

## Следующая лекция: Тестирование