Sessão 1 - 23/09/2025

Coordenador: Everton Vinicius
Quadro: João Gabriel
Mesa: Cláudio Daniel

Disciplina: **TEC502 - TP01 - Concorrência e Conectividade - 2025.2**

Problema: Jogo de Cartas Multiplayer Distribuído

Sessão: 1 — Apresentação do problema e levantamento inicial

Data: 23/09/2025

> Contexto resumido do problema: precisamos reengenheirar o protótipo do jogo para uma arquitetura distribuída com múltiplos servidores colaborando (escalabilidade, tolerância a falhas, consistência do estado e nova função de troca de cartas). A comunicação entre servidores será via API REST, e entre servidores e clientes via modelo publisher–subscriber. Tudo deve rodar em contêineres Docker e permitir duelos 1×1 entre servidores, com gestão distribuída de “pacotes de cartas” e testes para concorrência e falhas.

Fatos

- **Reengenharia para arquitetura distribuída**: vários servidores devem **hospedar partidas e dividir recursos** para crescer com a base de jogadores.
- **Eliminação do ponto único de falha** e **continuidade do serviço** mesmo com falhas de componentes.
- **Restrições técnicas**:
  - **Docker** para servidores e clientes (múltiplas instâncias/laboratório).
  - **Arquitetura descentralizada** em **contêineres separados** e **máquinas distintas**.
  - **Servidor ↔ Servidor: API REST** projetada pela equipe (testável com Insomnia/Postman).
  - **Servidor ↔ Clientes: publisher–subscriber** (permitido uso de bibliotecas).

- **Gestão distribuída** do **estoque global** de pacotes, garantindo **justiça e unicidade** sem centralização.
- **Partidas entre servidores** (pareamento 1×1 mantendo garantias do protótipo).
- **Tolerância a falhas** durante partidas/operações.
- **Teste de software** para **concorrência distribuída** e **cenários de falha**.

Ideias

- **Topologia geral:**
  - Múltiplos **Game Servers** rodando em Docker; cada um: salas/partidas locais, conexão com clientes (pub-sub) e API REST para cooperação inter-servidores (pareamento, troca de cartas, sincronizações pontuais).
- **Pub-Sub (Servidor–Clientes):**
  - **Tópicos** por sala/partida (ex.: `match/{id}`), lobby e eventos de inventário/troca; clientes **assinam** atualizações de estado e **publicam** ações do jogador.
- **REST (Servidor–Servidor):**
  - Endpoints idempotentes para **pareamento cross-servidor** (ex.: `/matches/handshake`, `/matches/{id}/state`), **sincronização pontual** de estado e **descoberta/heartbeat**.
- **Pareamento 1×1 entre servidores:**
  - **Anúncio** de filas locais via REST e **matchmaking bilateral** (handshake + confirmação); fallback para outro servidor se há falha no meio do processo.
- **Tolerância a falhas:**
  - **Heartbeats** entre servidores; **retransmissão** de eventos críticos; **commit em dois passos com timeout** e **retomada** de partidas via replays de eventos.

Questões

1. O que é uma API REST?
2. Para que serve o modelo publisher–subscriber?
3. Qual tecnologia usaremos para pub-sub (ex.: WebSockets, MQTT, NATS)?
4. O que é um “tópico/canal” no pub-sub e como vamos nomeá-los?
5. Como os clientes se autenticam/entram no jogo?
6. Como vamos identificar cada partida (ID da partida)?
7. Qual será o formato das mensagens (ex.: JSON) e quais campos mínimos precisam existir?
8. O que é idempotência e como evitamos processar a mesma ação duas vezes?
9. Como evitar que dois servidores entreguem o mesmo pacote de cartas para pessoas diferentes?
10. O que acontece se um servidor cair no meio da partida? Como retomamos?
11. Como os servidores “se encontram” (lista fixa, descoberta simples, heartbeat)?
12. Que logs básicos precisamos (ex.: `match_id`, `request_id`) para entender problemas?
13. O que é um sistema distribuído e como funciona?

Metas

- **Estudar e enquadrar o *Problema dos Generais Bizantinos*** para entender limites de consenso e como isso influencia a **consistência e a tolerância a falhas** no nosso cenário de jogo.
- **Pesquisar e aplicar o modelo *publisher–subscriber*** na comunicação **Servidor–Clientes**, definindo **tópicos**, **formato de mensagens**, **contratos de eventos** e uma **prova de conceito funcional** em Docker.
