Na sequencia tem a documentacao de um sistema. Ela foi sendo atualizada e acabou confunsa e redundante. Eu quero que vc me ajude a reorganizala de forma a que fique consisa e clara. Vc vai ver que tem workers. Todo o que seja geral a todos os workers, como por exemplo a dinamica da arbitragem (as operacoes t1, t2 e t3) que fique num lugar so. Depois, o que for particular de cada worker/exchange que seja documentado numa secao especifica para esse worker. Para começar vamos nos focar nos workers da bybit e da okx e no coordenador. Por enquanto nao vai printar codigo nenhuma. Me da um esboco basicao sem entrar em detalhes e a gente vai adaptando. É importante ter em mente que esta documentacao que sera produzida servira de base para que vc continue o desenvolvimento do sistema em chats futuros.

DOC:

Da uma olhada nesta doc e nos códigos e me diz brevemente o que vc entendeu. So olha os códigos coordenador.js, workerExecOKX e workerExecBybit.js. Os outros ignora por enquanto. Me diz brevemente o que haveria que fazer daqui em diante. Mas não faça código nenhum, so me responde.

**Projeto: Desenvolvimento de Sistema de Arbitragens em Tempo Real**

**(Versão Expandida 02/04/25)**

**Objetivo Geral**

Desenvolver um sistema baseado em worker\_threads do Node.js para detecção e execução de arbitragens triangulares de criptoativos em tempo real. O sistema opera com três níveis de Workers. O primeiro é o main, que controla os Workers de dados e o worker coordenador. O segundo são os Workers de dados e o coordenador, que controla os Workers de execução ( um para cada Exchange) das operações. O terceiro nível é o dos Workers executores, controlados pelo worker coordenador. A estrutura é projetada para escalar até 6-10 exchanges, garantindo modularidade, resiliência e baixa latência.

**Estrutura Geral**

**A comunicação segue a cadeia:**

* **main → workers de dados (dados de mercado).**
* **main → workerCoordenador (matriz de operações).**
* **workerCoordenador → workers executores (ordens específicas como T1, T2, T3).**
* **Main**:
  + **Função**:
    - Inicializa e controla os workers do nível 1 (dados e coordenador). Responsável por inicializar os workers de dados e o workerCoordenador.
    - Recebe dados de mercado dos workers de dados.
    - Detecta arbitragens com uma função simples (ex.: if t > a\*b/c + r, onde t é o preço alvo, a, b, c são pares, e r é o lucro mínimo).
    - Monta a matriz de operações (ex.: [T1: comprar BTC em OKX, T2: vender BTC em Bybit, T3: comprar USDT em Bybit]) e a envia ao workerCoordenador em uma única mensagem.
  + **Papel**: Ponto central de roteamento e detecção, sem gerenciar estados complexos.
* **Nível 1: Detecção e Coordenação (Dependente do Main)**
  + **Workers de Dados**: Um por exchange (ex.: workerDadosOKX, workerDadosBybit).
    - **Função**: Recebem dados de mercado (order books, preços) via WebSocket público e enviam ao main para análise.
    - **Dependência**: Controlados pelo main, que os inicializa e consome seus dados.
    - **Estado**: Conexões WebSocket abertas e ativas desde o início. Com estas informações o main avalia a possibilidade de arbitragem e caso detecte alguma envia as informações ao worker coordenador (matriz de operações) para sua execução.
  + **Worker Coordenador (workerCoordenador)**:
    - **Função**:
      * Recebe a matriz de operações do main.
      * Distribui ordens aos workers de execução segundo a seguinte lógica:
        + Envia T1 ao worker correspondente, (no ex.: T1 ao workerExecOKX) O worker posta a ordem e devolve o resultado.
        + Se T1 foi efetivamente executada: ‘filled’ , o coordenador envia T2 e T3 (se houver) aos workers correspondentes.
        + Cada worker loga o resultado destas operações e o devolve ao coordenador.
        + Se a T1 não for ‘filled’ o coordenador aborta a arbitragem. Isto é, avisa do fracasso e não dispara T2 e T3.
      * O coordenador loga resultados e a princípio não necessita notificar o main. Mais adiante precisara montar um log file com detalhes de toda a operativa.
      * É importante entender, observar e respeitar o mapeamento e ajuste dos nomes de campos entre a matriz de operações e as respectivas definições nas diferentes exchanges.
    - **Dependência**: Recebe instruções do main e as executa.
    - **Estado**: Conexões abertas e mantidas ociosas desde o inicio (com pings ex.: OKX: 25s, Bybit: 20s e reconectando após 5s se falharem) à espera de arbitragens. Rastreia cada arbitragem com orderId (ex.: t1OrderId para T1).
  + **Workers Opcionais (ex.: workerSaldos)**:
    - **Função**: Monitorar saldos e fluxos de caixa via WebSocket privado, enviando dados ao main para análise futura.
* **Nível 2: Execução (Dependente do Worker Coordenador)**
  + **Workers de Execução**: Um por exchange (ex.: workerExecOKX, workerExecBybit).
    - **Função**:
      * Gerenciam conexões WebSocket privadas para trades e ordens.
      * Executam ordens recebidas do workerCoordenador (ex.: T1: LIMIT IOC, T2/T3: MARKET ou LIMIT).
      * Retornam status ao coordenador (ex.: accepted, filled, rejected, cancelled).
    - **Dependência**: Controlados pelo workerCoordenador, que envia ordens e espera respostas.
    - **Estado**: Conexões WebSocket abertas no início, sinalizando "prontas" ao main ou coordenador, mantidas ociosas com pings.

**Desenvolvimento.**

* O main e os workers de dados já estão desenvolvidos, faltando so pequenas adaptações.
* A próxima etapa de desenvolvimento é a do worker coordenador e dos workers de execução.

O coordenador so vai administrar e coordenar a execução das operações T que chegam através da matriz de operacoes. A princípio a logica dele será bastante simples e pode ser extraída do workerTrades.js.

O worker de execução sera o encarregado de administrar a comunicação com o server. Para isso fara o seguinte:

1. Abre os websockets necessários (ex: para OKX é um, para Bybit são dois) e subscribe aos canais pertinentes. Depois disso fica esperando a eventual chegada de uma ordem de execução. Como a conexão fica ociosa por longos períodos ela tem que ser mantida viva através dos ping/pong.
2. Quando a chega uma ordem de execução do coordenador (T1, T2 ou T3) adequa os parâmetros recebidos aos formatos e estrutura da Exchange e posta o trade.
3. Espera as mensagens de resposta.
4. Quando recebe uma mensagem a reenvia ao coordenador que a analisa e resolve os passos seguintes.
5. A logica e formatos de comunicação com a exchange podem ser extraídos do workerConexoes.js e/ou do workerTrades.js.

* Os workers de execução serão desenvolvidos um por vez junto com sua integração ao coordenador. O primeiro a ser desenvolvido é o da OKX.

**Fluxo de Operação**

1. **Inicialização**:
   * O main inicia os workers de dados, o workerCoordenador e os workers de execução.
   * Workers de dados abrem WebSockets públicos e enviam dados de mercado ao main.
   * Workers de execução abrem WebSockets privados, autenticam (ex.: HMAC-SHA256) e sinalizam "prontos".
2. **Detecção**:
   * O main analisa dados de mercado, detecta uma arbitragem e monta a matriz de operações.
   * Envia a matriz ao workerCoordenador via MessageChannel.
   * A partir do momento da detecção começa o Rastreamento do Tempo (ver mais adiante), para analisar o tempo transcorrido durante toda a arbitragem. Este seria o momento t=0.
3. **Coordenação**:
   * O workerCoordenador recebe a matriz, envia T1 ao worker correspondente e monitora o status:
     + Se T1 é rejected ou cancelled, aborta e loga.
     + Se T1 é filled, envia T2 e T3 aos workers correspondentes.
   * Recebe status de T2/T3 e loga o resultado final.
   * Cada vez que uma mensagem é recebida ou enviada pelo worker, seja da plataforma ou do coordenador, o tempo transcorrido deverá ser informado, junto com o abs.
4. **Execução**:
   * Cada worker de execução formata a ordem (ex.: OKX: op: 'order', Bybit: op: 'order.create'), envia via WebSocket e retorna o status ao coordenador.

**Componentes Principais**

* **Main**:
  + **Inicialização**: Cria workers com new Worker, passando credenciais via workerData.
  + **Detecção**: Função simples que compara preços e monta a matriz (ex.: { exchange, par, side, type, amount, price, timeInForce }).
  + **Comunicação**: Envia { type: 'start', operacoes } ao workerCoordenador.
* **Workers de Dados**:
  + **Conexão**: WebSocket público (ex.: OKX: wss://ws.okx.com:8443/ws/v5/public, Bybit: wss://stream.bybit.com/v5/public).
  + **Parsing**: Extrai preços e profundidade do order book, enviando ao main em formato padronizado (ex.: { exchange, par, bid, ask }).
  + **Manutenção**: Pings regulares e reconexão automática.
* **Worker Coordenador**:
  + **Distribuição**: Mapeia ordens da matriz para workers de execução usando exchange como chave.
  + **Monitoramento**: Usa t1PendingOrderId (ID enviado) e t1OrderId (ID da exchange) para rastrear T1, com timeout de 3s.
  + **Decisões**: Aborta se T1 falhar, dispara T2/T3 se T1 for preenchida, logando com tempos relativos e absolutos.
* **Workers de Execução**:
  + **Conexão**:
    - OKX: Um WebSocket (wss://ws.okx.com:8443/ws/v5/private), autenticado com apiKey, passphrase, timestamp, HMAC-SHA256 de GET/users/self/verify.
    - Bybit: Dois WebSockets (wss://stream.bybit.com/v5/trade para ordens, wss://stream.bybit.com/v5/private para status), autenticados com apiKey, expires, HMAC-SHA256 de GET/realtime${expires}.
  + **Execução**:
    - Formata ordens (ex.: OKX: ordType: 'ioc', Bybit: timeInForce: 'IOC').
    - Envia via WebSocket e parseia respostas (ex.: OKX: state: 'filled', Bybit: orderStatus: 'Filled').
  + **Status**: Retorna ao coordenador com detalhes (ex.: avgPx, fillSz, cancelSource).

**Detalhes Técnicos**

* **Comunicação**:
  + **Main → Coordenador**: { type: 'start', operacoes: [{ exchange, par, side, type, amount, price, timeInForce }, ...] }.
  + **Coordenador → Execução**: { type: 'executeOrder', order: { exchange, par, side, type, amount, price, timeInForce }, orderId }.
  + **Execução → Coordenador**: { type: 'orderStatus', orderId, status, instId, avgPx, fillSz, cancelSource }.
  + Usa MessageChannel para baixa latência.
* **Timeout**: 3s para T1, abortando se não preenchida.
* **Conexões**:
  + Abertas no início, mantidas com pings (OKX: 25s, Bybit: 20s), reconectando após 5s.
  + Sinalizam "prontas" com { type: 'conexoesProntas' }.
* **Matriz de Operações**: Exemplo:

const operacoes = [

{ exchange: 'OKX', symbol: 'BTCUSDT', side: 'buy', type: 'limit', amount: '0.00001', price: '82000', timeInForce: 'ioc' },

{ exchange: 'OKX', symbol: 'BTCBRL', side: 'sell', type: 'market', amount: '0.0009', price: '488000', timeInForce: null },

{ exchange: 'OKX', symbol: 'USDTBRL', side: 'buy', type: 'market', amount: '1.00', price: null, timeInForce: null }

];

**Rastreamento do Tempo.**

(Fases: Ver ATUALIZAÇÃO DO DESENVOLVIMENTO NO 03/04/2025)

Como as ordens podem ir para exchanges diferentes (ex.: OKX, Bybit, Binance), e cada worker será independente, o rastreamento de tempo precisa ser consistente e centralizado no workerCoordenador. O workerExecOKXFase2 (e futuros workers) deve ser simples e apenas reportar eventos com timestamps absolutos (abs) e o tempo relativo (t) baseado no t0 recebido. Aqui está minha sugestão para ajustar isso:

Estrutura proposta:

1. mainFase2 (simulando o início):
   * Define t0 = Date.now() no momento da detecção da arbitragem.
   * Envia a matriz completa (T1, T2, T3) com t0 ao workerCoordenador (por enquanto, simulado enviando apenas T1 ao workerExecOKXFase2).
2. workerCoordenador (futuro, mas simulado por enquanto):
   * Recebe a matriz e t0.
   * Para cada operação (T1, T2, T3), envia a ordem ao worker correspondente (ex.: workerExecOKX, workerExecBybit) com t0 incluído na mensagem.
   * Monitora os status retornados pelos workers:
     + Calcula t = abs - t0 para cada evento (envio de ordem, recebimento de status).
     + Decide o próximo passo (abortar ou enviar T2/T3) com base no status de T1.
   * Loga todos os eventos com abs e t.
3. workerExecOKXFase2 (e futuros workers):
   * Recebe a ordem e t0 do coordenador.
   * Para cada evento (envio da ordem, recebimento de resposta da OKX), calcula t = abs - t0 e inclui nos logs e nas mensagens de status.
   * Não precisa saber se é T1, T2 ou T3, apenas processa o que chega e devolve o status com abs e t.
   * Funciona independentemente da exchange, então o mesmo modelo pode ser replicado para outras (ex.: workerExecBybit).

**Como lidar com múltiplas exchanges.**

* Cada worker (ex.: workerExecOKX, workerExecBybit) será uma instância separada, conectada à sua respectiva exchange.
* O workerCoordenador gerencia a comunicação com todos os workers via MessageChannel (um canal por worker).
* O t0 é único para a arbitragem inteira, definido no mainFase2 e passado a todos os workers via workerCoordenador. Assim, todos os tempos relativos (t) são consistentes, independentemente da exchange ou do worker.
* Exemplo de matriz com exchanges diferentes:

[

{ exchange: 'OKX', par: 'BTC-USDT', side: 'buy', ... },

{ exchange: 'Bybit', par: 'BTC-USDT', side: 'sell', ... },

{ exchange: 'Binance', par: 'USDT-BRL', side: 'buy', ... }

]

* + O workerCoordenador envia cada operação ao worker correto e rastreia os tempos de todos os eventos.

Sugestão para os ajustes no código:

1. mainFase2.js:
   * Adicionar t0 = Date.now() antes de montar a matriz.
   * Enviar { type: 'executeOrder', order, clientOrderId, t0 } ao workerExecOKXFase2 (simulando o envio ao coordenador por enquanto).
2. workerExecOKXFase2.js:
   * Armazenar t0 recebido na mensagem (ex.: em uma variável global ou no escopo do worker).
   * Modificar logMessage para calcular t = abs - t0 e incluir no log (ex.: [abs=12345 t=50]).
   * Adicionar t: abs - t0 às mensagens orderStatus enviadas ao parentPort.
   * Corrigir ws = new WebSocket(exchanges.OKX.wsUrl).
3. Futuro workerCoordenador (não implementado agora):
   * Receber a matriz e t0 do mainFase2.
   * Criar um MessageChannel para cada worker (OKX, Bybit, etc.).
   * Enviar ordens com t0 e rastrear t para cada evento, logando e decidindo o fluxo.

Se opta por manter o workerExecOKXFase2 simples e genérico, apenas processando ordens e reportando status com abs e t calculado a partir de t0. O workerCoordenador será o cérebro da operação, recebendo a matriz, distribuindo ordens para os workers corretos (mesmo que sejam exchanges diferentes) e rastreando todos os tempos relativos em um único lugar. Para esta Fase 2, isso se simula enviando uma ordem do mainFase2 ao workerExecOKXFase2 com t0, já preparando o terreno para a Fase 3.

**Dinâmica de Comunicação com a OKX: Canais de Postagem e Ordens.**

**(\*\* Todas as exchanges utilizam estruturas diferentes, mas suas dinâmicas são similares a esta.)**

A OKX utiliza dois canais distintos no WebSocket privado para gerenciar ordens, cada um com funções específicas e papéis bem definidos no processo de envio e execução de ordens:

1. Canal de Postagem (op: 'order')
   * Função: Responsável por receber a solicitação de postagem de uma ordem e validar seus aspectos técnicos antes de enviá-la ao mercado.
   * Comportamento:
     + Quando uma ordem é enviada (ex.: via ws.send() com op: 'order'), o canal avalia parâmetros como formato, saldo disponível e consistência técnica (ex.: amount > 0).
     + Aceitação (code: "0"): Se a validação técnica é bem-sucedida, retorna code: "0" e um ordId (ID da ordem na exchange), indicando que a ordem foi aceita e enviada ao mercado. Isso significa apenas que a ordem é tecnicamente válida, sem nenhuma garantia sobre sua execução no mercado.
     + Rejeição (code !== "0"): Se há uma inconsistência técnica (ex.: saldo insuficiente, amount=0), retorna um código de erro (ex.: 60011) e uma mensagem (ex.: Order amount too low). Nesse caso, a ordem não é postada no mercado, e o canal de ordens não recebe nenhuma atualização.
   * Respostas Possíveis:
     + Apenas accepted (validação técnica OK) ou rejected (erro técnico).
   * Exemplo de Resposta:
     + Aceita: {"op": "order", "code": "0", "data": [{"ordId": "12345", "instId": "BTC-USDT"}]}
     + Rejeitada: {"op": "order", "code": "60011", "msg": "Order amount too low"}
2. Canal de Ordens (orders)
   * Função: Reporta o estado da ordem no mercado após ela ser aceita e postada pelo canal de postagem.
   * Comportamento:
     + Ativado somente após a ordem ser validada e aceita pelo canal de postagem (code: "0").
     + Estado Inicial (live): Envia uma mensagem indicando que a ordem chegou ao mercado e está ativa no livro de ordens. Este estado pode ser reportado, mas não é um pré-requisito para ações subsequentes.
     + Ordens LIMIT Normais: Após live, a ordem permanece no livro de ordens indefinidamente até ser preenchida (filled) ou cancelada (cancelled) por ação externa (ex.: usuário ou timeout), mas essas atualizações não são garantidas ou imediatas.
     + Ordens IOC (Immediate Or Cancel): Após live, a plataforma avalia imediatamente se há contraparte no mercado para executar a ordem. O estado muda rapidamente para:
       - filled: Se a ordem é executada total ou parcialmente, com detalhes como avgPx (preço médio) e fillSz (quantidade preenchida).
       - cancelled: Se não há contraparte suficiente, com cancelSource indicando o motivo (ex.: IOC not filled).
   * Respostas Possíveis:
     + live (ordem no mercado), filled (executada), ou cancelled (não executada).
   * Exemplo de Resposta:
     + Live: {"arg": {"channel": "orders"}, "data": [{"ordId": "12345", "state": "live", "instId": "BTC-USDT"}]}
     + IOC Filled: {"arg": {"channel": "orders"}, "data": [{"ordId": "12345", "state": "filled", "avgPx": "82700", "fillSz": "0.00001"}]}
     + IOC Cancelled: {"arg": {"channel": "orders"}, "data": [{"ordId": "12345", "state": "cancelled", "cancelSource": "IOC not filled"}]}
3. Interação entre os Canais
   * O canal de postagem atua como um "porteiro" técnico: se a ordem não passa na validação (rejected), o processo termina ali, e o canal de ordens não é acionado. Se passa (accepted), o canal de ordens reporta o estado no mercado conforme as mensagens chegam.
   * As mensagens são assíncronas, e o coordenador processa o que recebe sem esperar um estado específico como live. O ordId é usado para correlacionar eventos entre os canais.
   * Para ordens IOC, o fluxo típico é: accepted (postagem) → live (ordens, opcional) → filled ou cancelled (ordens), em sequência rápida.
4. Implicações para o Worker Coordenador
   * Rejeição na Postagem: Se o canal de postagem retorna rejected, o coordenador loga o motivo (errorCode, errorMsg) e aborta a arbitragem imediatamente. Nenhuma mensagem do canal de ordens é esperada ou processada.
   * Aceitação e Mercado: Após accepted, o coordenador ativa um timeout de 3 segundos ao enviar T1 e reage às mensagens do canal de ordens conforme chegam:
     + live: Loga que a ordem está no mercado e mantém o timeout ativo. Se nada mais chegar em 3 segundos, o timeout aborta a arbitragem.
     + filled: Prossegue com T2/T3, logando avgPx e fillSz, e cancela o timeout.
     + cancelled: Aborta, logando cancelSource.
   * Timeout: Se nenhuma mensagem de filled chegar dentro de 3 segundos após o envio de T1, o coordenador aborta a arbitragem, assumindo que a ordem não foi executada a tempo.
   * Aborto: Após rejected (postagem), cancelled (ordens), ou timeout, o coordenador seta um flag (aborted) para ignorar mensagens futuras, garantindo que T2/T3 não sejam disparados.

Essa dinâmica reflete o comportamento real da OKX, onde o canal de postagem valida tecnicamente a ordem e o canal de ordens reporta seu estado no mercado, com o coordenador reagindo dinamicamente às mensagens recebidas e usando um timeout como salvaguarda.

**Adaptação de Parâmetros de Ordem para a OKX:**Na matriz de operações enviada pelo workerCoordenador, as ordens são definidas com os campos type (ex.: 'limit', 'market') e timeInForce (ex.: 'IOC' ou null), que são padrões comuns em exchanges. Porém, a OKX utiliza um único campo, ordType, que combina essas informações. Para compatibilidade, o workerExecOKX deve mapear os valores da seguinte forma:

* Se type: 'limit' e timeInForce: 'IOC', então ordType: 'ioc'.
* Se type: 'limit' e timeInForce: null, então ordType: 'limit'.
* Se type: 'market' e timeInForce: null, então ordType: 'market'.  
  Essa conversão deve ser implementada na função formatOrder do workerExecOKX para garantir que as ordens sejam postadas corretamente no formato esperado pela OKX (ex.: { op: 'order', args: [{ instId, side, ordType, sz, px, ... }] })."

Alem disso,

O erro comum ao configurar ordens, o 51020 ("Your order should meet or exceed the minimum order amount") da OKX, ocorre quando o tamanho da ordem (sz) não atende ao mínimo exigido pela exchange. Isso pode acontecer por:

* **Unidade errada do sz**: Em ordens a mercado, o sz pode ser interpretado em BTC (moeda base) ou USDT (moeda de cotação), dependendo do tgtCcy. Se configurado incorretamente (ex.: tgtCcy: "quote\_ccy" com sz em BTC), o valor fica muito baixo (ex.: 0.0007 USDT em vez de 0.0007 BTC).
* **Mínimos implícitos**: Algumas exchanges impõem um valor mínimo em moeda de cotação (ex.: $5-$10 USDT) para ordens a mercado, além do minSz em moeda base (ex.: 0.00001 BTC).
* **Outras exchanges**: Esse problema pode aparecer em plataformas como Binance, Bybit ou KuCoin, onde ordens a mercado também podem exigir um valor mínimo em USDT ou outra moeda fiat/stablecoin, especialmente para compras (buy). Sempre cheque a documentação ou API de instrumentos para confirmar os mínimos (minSz, minNotional, etc.).

**Solução**: Especifique claramente a unidade do sz (BTC ou USDT) e ajuste o valor para atender aos mínimos da exchange.

**Regra Simples**

Assumindo trades em modo spot (tdMode: "cash") e tamanhos sempre em BTC:

* **Limite (buy ou sell)**: sz em BTC, px obrigatório, sem tgtCcy (não precisa, melhor omitir).
* **Mercado (buy ou sell)**: sz em BTC, use tgtCcy: "base\_ccy" (melhor incluir), sem px.

**Tamanho e Minimo dos Lotes**

Resumo dos Mínimos Confirmados:

| Par | Moeda Base | minSz (BTC) |  |
| --- | --- | --- | --- |
| BTC-USDT | BTC | 0.00001 |  |
| BTC-BRL | BTC | 0.0001 | 1/10 USDT |
| USDT-BRL | USDT | USDT 1) |  |

Os tick sizes são diferentes também.

**Parâmetros Principais**

* **instId**: Identificador do instrumento (ex.: "BTC-USDT").
* **side**: Lado da ordem:
  + "buy": Compra.
  + "sell": Venda.
* **ordType**: Tipo de ordem:
  + "limit": Ordem limite, executada a um preço específico (px).
  + "market": Ordem a mercado, executada imediatamente ao preço atual.
* **sz**: Tamanho da ordem (quantidade a negociar).
  + Em BTC para todos os casos aqui (conforme a regra).
* **px**: Preço (obrigatório para limit, ignorado para market).
* **tdMode**: Modo de trading:
  + "cash": Spot (sem margem), usado aqui.
  + "cross": Margem cruzada.
  + "isolated": Margem isolada.
* **tgtCcy**: Moeda alvo do sz (aplicável apenas a ordens a mercado):
  + "base\_ccy": sz em BTC.
  + "quote\_ccy": sz em USDT.
  + Padrão: Pode variar; especifique para evitar erros.
* **ccy**: Moeda da margem (ex.: "USDT"), opcional para spot.

**Requisitos Mínimos (BTC-USDT, Spot)**

* **minSz: "0.00001" BTC**: Mínimo em BTC para spot (OKX).
* **Mínimo em USDT (implícito)**: ~$5-$10 para ordens a mercado com tgtCcy: "quote\_ccy".

Veja mais detalhes no ‘Guia Completo para Configurar Ordens na OKX via WebSocket’ e em https://www.okx.com/docs-v5/en/?shell#order-book-trading-trade-ws-place-order

**Observações para o Futuro**

1. **Exclusão de Exchanges Ocupadas**: "Futuramente, o main ou workerCoordenador deve excluir exchanges com arbitragens em andamento das avaliações de novas arbitragens."
2. **Saldos Insuficientes para T2/T3**: "Saldo insuficiente para T2/T3 será logado como aviso, sem ação imediata, já que T1 captura o lucro."
3. **Múltiplas Arbitragens**: "Gerenciar múltiplas arbitragens será resolvido futuramente, atrelado à exclusão de exchanges ocupadas. Por ora, assume-se raridade de sobreposição."

**Integração com main e Desenvolvimento dos workers Coordenador e Execuções.**

A integração entre o main e o coordenador e seus workers sera desenvolvida mais adiante. De momento sera simulada, o coordenador iniciara com a matriz de operações como se esta houvesse sido recebida do main.

**Códigos Para Reutilização.**

**workerConexoes.js**

const { parentPort, workerData } = require('worker\_threads');

const WebSocket = require('ws');

const crypto = require('crypto');

// Cores ANSI

const GREEN = '\x1b[32m';

const BROWN = '\x1b[33m';

const YELLOW = '\x1b[93m';

const RESET = '\x1b[0m';

// Credenciais da OKX (via workerData)

const { okxApiKey, okxApiSecret, okxPassphrase } = workerData;

// Configuração modular das exchanges

const exchanges = {

  OKX: {

    wsUrl: 'wss://ws.okx.com:8443/ws/v5/private',

    formatOrder: (order) => {

      const now = new Date();

      const hhmmssSSS = now.toISOString().slice(11, 23).replace(/[:.]/g, '');

      return {

        id: `ORD${hhmmssSSS}`,

        op: 'order',

        args: [{

          instId: order.par,

          side: order.side,

          ordType: order.timeInForce === 'IOC' ? 'ioc' : order.type,

          sz: order.amount,

          px: order.price || undefined,

          tdMode: 'cash',

          ccy: 'USDT'

        }]

      };

    },

    parseStatus: (msg, channel) => {

      if (channel === 'trade') {

        return msg.code === '0' ? 'accepted' : 'rejected';

      } else if (channel === 'orders') {

        return msg.state === 'filled' ? 'filled' : msg.state === 'canceled' ? 'cancelled' : 'accepted';

      }

    }

  }

};

/\*\*

 \* Dinâmica das mensagens da OKX e outras exchanges:

 \*

 \* A OKX usa dois fluxos principais no WebSocket privado:

 \* 1. Resposta de postagem (`op: "order"`):

 \*    - Confirma se a ordem foi recebida pela exchange.

 \*    - `code: "0"`: Ordem aceita, registrada com um `ordId`. O estado real vem do canal `orders`.

 \*    - `code !== "0"`: Ordem rejeitada (ex.: saldo insuficiente, parâmetros inválidos). Nesse caso,

 \*      o canal `orders` não envia atualizações, pois a ordem não foi criada.

 \*

 \* 2. Canal `orders`:

 \*    - Fornece atualizações em tempo real do estado da ordem após ser aceita.

 \*    - Para ordens `limit` (não IOC):

 \*      - `state: "live"`: Ordem aceita e no livro de ordens. Não há mais mensagens até

 \*        `filled` (preenchida) ou `canceled` (cancelada), o que pode nunca ocorrer.

 \*    - Para ordens `market`:

 \*      - `state: "live"`: Ordem aceita, seguida rapidamente por `state: "filled"` com o preço

 \*        de execução (ex.: `avgPx`, `fillPx`).

 \*    - Para ordens `IOC` (Immediate Or Cancel):

 \*      - `state: "live"`: Ordem aceita, seguida por:

 \*        - `state: "filled"`: Preenchida total/parcialmente, com preço (ex.: `avgPx`).

 \*        - `state: "canceled"`: Não preenchida, com motivo (ex.: `cancelSource`).

 \*

 \* Por que essa dinâmica?

 \* - A separação entre confirmação de postagem e atualizações de estado é comum em exchanges

 \*   (ex.: Binance, Bybit) para garantir baixa latência nas atualizações (canal `orders`) e

 \*   feedback imediato sobre erros de postagem. A ordem das mensagens é assíncrona e depende

 \*   de latência/processamento, então o código deve ser robusto para qualquer sequência.

 \*/

function formatObject(obj) {

  return JSON.stringify(obj, null, 2)

    .replace(/"([^"]+)": "([^"]+)"/g, `"$1": ${GREEN}"$2"${RESET}`)

    .replace(/"([^"]+)": (\d+(.\d+)?)/g, `"$1": ${BROWN}$2${RESET}`)

    .replace(/"([^"]+)": null/g, `"$1": null`);

}

function logMessage(channel, message) {

  const absoluteTime = Date.now();

  console.log(`[workerConexoes - ${channel}] [abs=${absoluteTime}] ${message}`);

}

// Mapa para rastrear ordens postadas e suas confirmações

const postedOrders = new Map();

function generateOKXSignature(timestamp, method, path, body) {

  const prehashString = timestamp + method + path + (body || '');

  return crypto.createHmac('sha256', okxApiSecret).update(prehashString).digest('base64');

}

let ws;

let portToTrades;

let pingInterval;

function connectOKXWebSocket() {

  ws = new WebSocket(exchanges.OKX.wsUrl);

  ws.on('open', () => {

    logMessage('Conexão', 'Conectado ao WebSocket OKX.');

    const timestamp = (Date.now() / 1000).toString();

    const sign = generateOKXSignature(timestamp, 'GET', '/users/self/verify', '');

    ws.send(JSON.stringify({

      op: 'login',

      args: [{

        apiKey: okxApiKey,

        passphrase: okxPassphrase,

        timestamp: timestamp,

        sign: sign

      }]

    }));

  });

  ws.on('message', (data) => {

    if (data.toString() === 'pong') {

      logMessage('Conexão', 'Pong recebido.');

      return;

    }

    const msg = JSON.parse(data);

    logMessage('Geral', `Mensagem recebida:\n${formatObject(msg)}`);

    if (msg.event === 'login' && msg.code === '0') {

      logMessage('Conexão', 'Autenticado com sucesso.');

      ws.send(JSON.stringify({

        op: 'subscribe',

        args: [

          { channel: 'orders', instType: 'SPOT', instId: 'BTC-USDT' },

          { channel: 'orders', instType: 'SPOT', instId: 'BTC-BRL' },

          { channel: 'orders', instType: 'SPOT', instId: 'USDT-BRL' }

        ]

      }));

      setTimeout(() => parentPort.postMessage({ type: 'conexoesProntas' }), 1000);

      pingInterval = setInterval(() => {

        if (ws.readyState === WebSocket.OPEN) {

          ws.send('ping');

          logMessage('Conexão', 'Ping enviado.');

        }

      }, 25000);

    }

    if (msg.event === 'subscribe' && msg.arg?.channel === 'orders') {

      logMessage('Conexão', `Inscrito no canal orders para ${msg.arg.instId}.`);

    }

    if (msg.op === 'order') {

      if (msg.code === '0') {

        if (msg.data && Array.isArray(msg.data) && msg.data.length > 0) {

          const orderData = msg.data[0];

          postedOrders.set(orderData.ordId, Date.now());

          logMessage('Postagem', `Confirmação de postagem recebida para ordem ${orderData.ordId}.`);

          // Não envia "accepted" aqui, deixa o canal orders cuidar disso

        } else {

          logMessage('Postagem', 'Resposta de ordem aceita sem dados válidos.');

        }

      } else {

        // Envia "rejected" para o workerTrades se a postagem falhar

        const instId = msg.args && Array.isArray(msg.args) && msg.args[0] ? msg.args[0].instId : 'unknown';

        logMessage('Postagem', `Ordem rejeitada: ${msg.msg}`);

        portToTrades.postMessage({

          type: 'orderStatus',

          exchange: 'OKX',

          orderId: msg.id,

          status: 'rejected',

          errorCode: msg.code,

          errorMsg: msg.msg,

          instId: instId

        });

      }

    }

    if (msg.arg?.channel === 'orders') {

      logMessage('Orders', `Mensagem bruta do canal orders:\n${JSON.stringify(msg, null, 2)}`);

      if (msg.data && Array.isArray(msg.data) && msg.data.length > 0) {

        msg.data.forEach(order => {

          const orderId = order.ordId;

          if (!postedOrders.has(orderId)) {

            logMessage('Orders', `${YELLOW}Warning: Atualização do canal orders (${order.state}) recebida antes da confirmação de postagem para ordem ${orderId}.${RESET}`);

          }

          const status = exchanges.OKX.parseStatus(order, 'orders');

          portToTrades.postMessage({

            type: 'orderStatus',

            exchange: 'OKX',

            orderId: orderId,

            status: status,

            instId: order.instId,

            // Detalhes adicionais para o workerTrades

            avgPx: order.avgPx || null,         // Preço médio de execução (para "filled")

            fillSz: order.fillSz || null,       // Quantidade preenchida (para "filled")

            cancelSource: order.cancelSource || null // Motivo do cancelamento (para "cancelled")

          });

        });

      } else {

        logMessage('Orders', 'Mensagem do canal orders recebida sem dados válidos.');

      }

    }

  });

  ws.on('close', () => {

    logMessage('Conexão', 'WebSocket desconectado. Tentando reconectar em 5s...');

    clearInterval(pingInterval);

    setTimeout(connectOKXWebSocket, 5000);

  });

  ws.on('error', (err) => {

    logMessage('Conexão', `Erro no WebSocket: ${err.message}`);

  });

}

function sendOrder(exchange, order) {

  const formattedOrder = exchanges[exchange].formatOrder(order);

  logMessage('Postagem', `Enviando ordem:\n${formatObject(formattedOrder)}`);

  ws.send(JSON.stringify(formattedOrder));

  // Novo: notifica o workerTrades sobre o ID da ordem enviada

  portToTrades.postMessage({

    type: 'orderPosted',

    orderId: formattedOrder.id,

    exchange: exchange

  });

}

parentPort.on('message', (msg) => {

  if (msg.type === 'init') {

    portToTrades = msg.portToTrades;

    connectOKXWebSocket();

    portToTrades.on('message', (msg) => {

      if (msg.type === 'postOrder') {

        logMessage('Postagem', `Recebida ordem do trades:\n${formatObject(msg)}`);

        sendOrder(msg.exchange, msg.order);

      }

    });

  }

});

**ATUALIZAÇÃO DO DESENVOLVIMENTO NO 03/04/2025.**

Foi iniciada a etapa de codificação do worker coordenador. Foi criado o código mainCorda.js mas, ainda não interage diretamente com o worker de execuções da OKX, senão com uma versão simulada dele. A próxima etapa consistirá na elaboração do workerExecOKX real, com base nos Codigos Para Reutilizacao. A Etapa se divide em três Fases.

* Fase 1. Produzira apenas o código que conecta com a Exchange e subscreve os canais.
* Fase 2. Acrescentará ao código da Fase 1 a postagem de uma operação enviada por um coordenador simulado.
* Fase 3. Integração do código da Fase 2 ao coordenador real.

**Código Atual do Coordenador (mainCorda.js).**

const { MessageChannel } = require('worker\_threads');

// Cores ANSI para logs

const GREEN = '\x1b[32m';

const BROWN = '\x1b[33m';

const RESET = '\x1b[0m';

// Matriz de operações fornecida

const operacoesSimuladas = [

  { exchange: 'OKX', par: 'BTC-USDT', side: 'buy', type: 'limit', amount: '0.00001', price: '82700', timeInForce: 'IOC' },

  { exchange: 'OKX', par: 'BTC-BRL', side: 'sell', type: 'limit', amount: '0.00001', price: '50000', timeInForce: null },

  { exchange: 'OKX', par: 'USDT-BRL', side: 'sell', type: 'limit', amount: '0.05', price: '6', timeInForce: null },

];

// Variáveis de estado

let operacoes = [];

let t1StartTime = null;

let t1Filled = false;

let t1OrderId = null;

let t1PendingOrderId = null; // ID da ordem enviada

let timeoutId = null;

let portToExecOKX; // Porta simulada para o workerExecOKX

let aborted = false; // Flag para indicar que a arbitragem foi abortada

// Função para formatar objetos nos logs

function formatObject(obj) {

  return JSON.stringify(obj, null, 2)

    .replace(/"([^"]+)": "([^"]+)"/g, `"$1": ${GREEN}"$2"${RESET}`)

    .replace(/"([^"]+)": (\d+(.\d+)?)/g, `"$1": ${BROWN}$2${RESET}`)

    .replace(/"([^"]+)": null/g, `"$1": null`);

}

// Função de log com timestamps

function logMessage(channel, message) {

  const absoluteTime = Date.now();

  const elapsed = t1StartTime ? `[t=${absoluteTime - t1StartTime}ms]` : '[t=0ms]';

  console.log(`[workerCoordenador - ${channel}] ${elapsed} [abs=${absoluteTime}] ${message}`);

}

// Função para enviar ordem ao workerExecOKX

function sendOrderToExec(order) {

  const orderId = `ORD${Date.now()}`; // ID único simulado

  logMessage('Postagem', `Enviando ordem para workerExecOKX (${order.par}):\n${formatObject(order)}`);

  portToExecOKX.postMessage({ type: 'executeOrder', order: order, orderId: orderId });

  return orderId;

}

// Simulação de inicialização e comunicação

function initialize() {

  const { port1, port2 } = new MessageChannel();

  portToExecOKX = port1;

  logMessage('Geral', 'Inicializando workerCoordenador.');

  // Simula o workerExecOKX respondendo

  portToExecOKX.on('message', (msg) => {

    if (msg.type === 'orderStatus' && !aborted) {

      const channel = msg.channel || 'Geral'; // Identifica o canal

      logMessage(channel, `Status recebido do workerExecOKX:\n${formatObject(msg)}`);

      // Canal de Postagem

      if (msg.orderId === t1PendingOrderId && msg.channel === 'Postagem') {

        if (msg.status === 'rejected') {

          logMessage('Postagem', `T1 rejeitada na postagem (par: ${msg.instId || operacoes[0].par}). Detalhes do erro: code=${msg.errorCode || 'N/A'}, msg=${msg.errorMsg || 'N/A'}. Abortando.`);

          clearTimeout(timeoutId);

          operacoes = [];

          t1PendingOrderId = null;

          aborted = true;

          return;

        } else if (msg.status === 'accepted') {

          t1OrderId = msg.orderId;

          logMessage('Postagem', `T1 aceita pelo workerExecOKX (par: ${msg.instId || operacoes[0].par}). Aguardando atualizações do mercado...`);

        }

      }

      // Canal de Ordens

      if (msg.orderId === t1OrderId && msg.channel === 'Ordens') {

        if (msg.status === 'live') {

          logMessage('Ordens', `T1 está no mercado (par: ${msg.instId || operacoes[0].par}).`);

        } else if (msg.status === 'filled') {

          t1Filled = true;

          logMessage('Ordens', `T1 preenchida (par: ${msg.instId || operacoes[0].par}). Detalhes: avgPx=${msg.avgPx}, fillSz=${msg.fillSz}. Disparando T2 e T3...`);

          clearTimeout(timeoutId);

          if (operacoes[1]) sendOrderToExec(operacoes[1]);

          if (operacoes[2]) sendOrderToExec(operacoes[2]);

        } else if (msg.status === 'cancelled') {

          logMessage('Ordens', `T1 cancelada (par: ${msg.instId || operacoes[0].par}). Detalhes: cancelSource=${msg.cancelSource || 'N/A'}. Abortando.`);

          clearTimeout(timeoutId);

          operacoes = [];

          t1PendingOrderId = null;

          aborted = true;

        }

      } else if (msg.orderId !== t1OrderId && msg.channel === 'Ordens') {

        logMessage('Ordens', `Status de T2/T3 recebido: ${msg.status} (par: ${msg.instId || 'desconhecido'})`);

      }

    }

  });

  // Simulação de respostas do workerExecOKX (port2) com dois canais

  port2.on('message', (msg) => {

    if (msg.type === 'executeOrder') {

      const { orderId, order } = msg;

      // Canal de Postagem (op: 'order')

      setTimeout(() => {

        if (Math.random() < 0.2) { // 20% de chance de rejeição técnica

          port2.postMessage({

            type: 'orderStatus',

            channel: 'Postagem',

            orderId,

            status: 'rejected',

            instId: order.par,

            errorCode: '60011',

            errorMsg: 'Order amount too low'

          });

        } else {

          port2.postMessage({

            type: 'orderStatus',

            channel: 'Postagem',

            orderId,

            status: 'accepted',

            instId: order.par

          });

          // Canal de Ordens (orders) só é ativado se aceita

          setTimeout(() => {

            port2.postMessage({

              type: 'orderStatus',

              channel: 'Ordens',

              orderId,

              status: 'live',

              instId: order.par

            });

            if (order.timeInForce === 'IOC') {

              setTimeout(() => {

                const status = Math.random() < 0.8 ? 'filled' : 'cancelled';

                port2.postMessage({

                  type: 'orderStatus',

                  channel: 'Ordens',

                  orderId,

                  status,

                  instId: order.par,

                  avgPx: status === 'filled' ? order.price : null,

                  fillSz: status === 'filled' ? order.amount : null,

                  cancelSource: status === 'cancelled' ? 'IOC not filled' : null

                });

              }, 200); // IOC resolve rapidamente após 'live'

            }

          }, 500); // 'live' vem após 'accepted'

        }

      }, 300); // Postagem responde primeiro

    }

  });

  // Simula recebimento da matriz do main

  setTimeout(() => {

    logMessage('Geral', 'Simulando recebimento da matriz de operações do main.');

    startArbitrage(operacoesSimuladas);

  }, 1000);

}

// Função para iniciar a arbitragem

function startArbitrage(ops) {

  operacoes = ops;

  aborted = false;

  logMessage('Geral', 'Iniciando arbitragem com matriz de operações.');

  if (operacoes[0]) {

    t1StartTime = Date.now();

    t1PendingOrderId = sendOrderToExec(operacoes[0]);

    timeoutId = setTimeout(() => {

      if (!t1Filled && operacoes.length > 0) {

        logMessage('Geral', 'Timeout de 3s para T1. Abortando.');

        operacoes = [];

        t1PendingOrderId = null;

        aborted = true;

      }

    }, 3000);

  } else {

    logMessage('Geral', 'Nenhuma operação T1 encontrada. Arbitragem não iniciada.');

  }

}

// Inicializa o worker

initialize();

**ATUALIZAÇÃO DO DESENVOLVIMENTO NO 06/04/2025.**

1. **Correção do id da Ordem**:
   * Resolvemos o erro 51000 ("Parameter id error") da OKX, causado por um id fixo (ORD\_TEST\_001). Implementamos um id dinâmico (ORD${hhmmssSSS}) gerado no formatOrder e introduzimos o clientOrderId como identificador interno dinâmico (CLI${hhmmssSSS}), mapeando-os com orderIdMap.
2. **Rastreamento de Tempo**:
   * Adicionamos o conceito de t0 (momento da detecção da arbitragem) no mainFase2 e o cálculo do tempo transcorrido (t = abs - t0) no workerExecOKXFase2. Isso foi incluído em logs (ex.: [abs=12345 t=50]) e mensagens de status, preparando o terreno para monitoramento de latência na arbitragem.
3. **Suporte a Múltiplas Exchanges, Coordenação e nomenclatura**:
   * Planejamos a estrutura para o futuro workerCoordenador, que gerenciará T1, T2 e T3, enviando ordens a workers específicos (ex.: OKX, Bybit) com base no status de T1 (filled, rejected, cancelled). O workerExecOKXFase2 foi mantido agnóstico, processando qualquer ordem recebida.
   * Nomenclatura de Campos. Para criar um sistema de trading de criptoativos em Node utilizando APIs e WebSockets, é essencial padronizar a nomenclatura dos parâmetros de trade entre as diferentes exchanges, já que cada uma possui variações em seus formatos e requisitos. Segue sintaxe correta dos parâmetros solicitados (symbol/par, side, type, amount/quantidade e timeInForce) para cada uma das exchanges mencionadas: OKX, Bybit, Binance, KuCoin, BitPreço, NovaDAX, Foxbit e Mercado Bitcoin. As informações são baseadas nas documentações oficiais das APIs de cada exchange, conforme disponíveis até a data atual (6 de abril de 2025), e refletem os padrões mais comuns para ordens de mercado e limite.
   * const operacoes = [
   * { exchange: 'OKX', symbol: 'BTCUSDT', side: 'buy', type: 'limit', amount: '0.00001', price: '82000', timeInForce: 'ioc' },
   * { exchange: 'OKX', symbol: 'BTCBRL', side: 'sell', type: 'market', amount: '0.0009', price: '488000', timeInForce: null },
   * { exchange: 'OKX', symbol: 'USDTBRL', side: 'buy', type: 'market', amount: '1.00', price: null, timeInForce: null }
   * ];
   * **Resumo Comparativo da Nomenclatura por Exchange**

| **Exchange** | **Symbol** | **Side** | **Type** | **Quantity** | **TimeInForce** |
| --- | --- | --- | --- | --- | --- |
| OKX | BTC-USDT | buy/sell | market/limit | sz | ioc/fok (via ordType) |
| Bybit | BTCUSDT | Buy/Sell | Market/Limit | qty | GTC/IOC/FOK |
| Binance | BTCUSDT | BUY/SELL | MARKET/LIMIT | quantity | GTC/IOC/FOK |
| KuCoin | BTC-USDT | buy/sell | market/limit | size | GTC/IOC/FOK |
| BitPreço | BTC-BRL | buy/sell | market/limit | amount | Não especificado (GTC) |
| NovaDAX | BTC\_BRL | BUY/SELL | MARKET/LIMIT | amount | Não especificado (GTC) |
| Foxbit | BTC\_BRL | buy/sell | market/limit | quantity | Não especificado (GTC) |
| Mercado Bitcoin | BTCBRL | buy/sell | market/limit | quantity | Não especificado (GTC) |

**Seção: Nomenclatura e Mapeamento de Campos para Exchanges**

**Introdução**

A matriz de operações, gerada pelo main e passada ao workerCoordenador e posteriormente aos workerExec (ex.: workerExecOKX), contém campos que descrevem cada ordem de arbitragem (ex.: comprar BTC, vender BTC, comprar USDT). Esses campos precisam ser mapeados para os parâmetros específicos das APIs das exchanges (como a OKX), que variam em nomenclatura e formatação. Esta seção detalha os campos da matriz, os ajustes necessários para a OKX, e considerações para expansão futura a outras exchanges.

**Campos da Matriz de Operações**

A matriz de operações utiliza uma nomenclatura genérica, projetada para ser flexível entre diferentes exchanges. Um exemplo típico é:

{ exchange: 'OKX', symbol: 'BTCUSDT', side: 'buy', type: 'limit', amount: '0.00001', price: '82000', timeInForce: 'ioc' },

{ exchange: 'OKX', symbol: 'BTCBRL', side: 'sell', type: 'market', amount: '0.0009', price: '488000', timeInForce: null },

{ exchange: 'OKX', symbol: 'USDTBRL', side: 'buy', type: 'market', amount: '1.00', price: null, timeInForce: null },

];

Os campos são:

* **exchange**: Identificador da exchange (ex.: OKX, Bybit, Binance). Usado internamente para rotear a ordem ao workerExec correspondente.
* **symbol**: Símbolo do par de negociação (ex.: BTCUSDT). Representa o instrumento financeiro, mas pode exigir ajustes de formatação por exchange.
* **side**: Lado da ordem (buy para compra, sell para venda).
* **type**: Tipo básico da ordem (limit para limite, market para mercado).
* **amount**: Quantidade a negociar (em unidade da moeda base ou cotada, dependendo do tipo de ordem e da exchange).
* **price**: Preço da ordem (obrigatório para limit, opcional ou ignorado para market).
* **timeInForce**: Política de execução temporal (ioc para "Immediate or Cancel", ou null para padrão da exchange).

Essa nomenclatura genérica permite que o main detecte oportunidades de arbitragem sem depender de detalhes específicos de cada exchange. No entanto, os workerExec precisam mapear esses campos para os parâmetros da API da exchange alvo.

**Mapeamento para a OKX**

A OKX, via sua API WebSocket privada (referência: <https://www.okx.com/docs-v5/en/#order-book-trading-trade-ws-place-order>), utiliza uma nomenclatura específica para ordens e subscrições. Os campos da matriz devem ser ajustados como segue:

 **symbol → instId**:

* + A OKX exige que o instId inclua um hífen (ex.: BTC-USDT, BTC-BRL, USDT-BRL).
  + A matriz usa o formato sem hífen (ex.: BTCUSDT), então o workerExecOKX deve incluir uma função como adjustSymbolForOKX para adicionar o hífen antes de enviar ordens ou subscrever canais WebSocket.
  + Exemplo: BTCUSDT → BTC-USDT.

 **side**:

* Compatível diretamente (buy ou sell).

 **type e timeInForce → ordType**:

* A OKX usa ordType para definir o tipo de ordem. Exemplos:
  + type: 'limit', timeInForce: 'ioc' → ordType: 'ioc' (ordem limite imediata ou cancelada).
  + type: 'limit', timeInForce: null → ordType: 'limit' (ordem limite padrão).
  + type: 'market', timeInForce: null → ordType: 'market' (ordem a mercado).
* O workerExecOKX já faz essa combinação na função formatOrder.

 **amount → sz**:

* Representa a quantidade a negociar (ex.: 0.00001 BTC ou 1.00 USDT).
* Na OKX, sz é o campo correspondente. O workerExecOKX deve mapear amount para sz.
* Para ordens a mercado, pode ser necessário especificar tgtCcy (ex.: base\_ccy para moeda base como BTC, quote\_ccy para moeda cotada como USDT).

 **price → px**:

* Preço da ordem (ex.: 82000 para USDT).
* Na OKX, px é o campo correspondente. O workerExecOKX deve mapear price para px.
* Ignorado em ordens market.

999

 **pcampo tgtCcy**

Na OKX deve ser considerado este campo. Assumindo trades em modo spot (tdMode: "cash") e tamanhos sempre em BTC, para operacoes:

**limit (buy ou sell)**: sz em BTC, px obrigatório, sem tgtCcy (não precisa, melhor omitir).

**market (buy ou sell)**: sz em BTC, use tgtCcy: "base\_ccy" (melhor incluir), sem px.

Por mais detalhes veja ANEXO REFERENCIAS EXEMPLOS CODIGOS ETC.DOC

1. **Ajuste para Trading Spot**:
   * Resolvemos o erro 51020 ("Your order should meet or exceed the minimum order amount") em ordens a mercado, adicionando tgtCcy: 'base\_ccy' para especificar sz em BTC e aumentando o amount de 0.0001 para 0.0007 BTC, atendendo aos requisitos mínimos da OKX em modo spot (tdMode: 'cash').

**Estado Atual:**

* O código está funcionando para ordens spot na OKX, com uma ordem a mercado (buy BTC-USDT) sendo enviada e processada corretamente.
* Suporta rastreamento de tempo, clientOrderId dinâmico, e está preparado para expansão para margin trading (tdMode: 'cross') ou outras exchanges com ajustes futuros.
* A base para o workerCoordenador foi planejada, mas ainda não implementada, pois estamos na Fase 2.

**Codigos no Estado Atual:**

**mainFase2.js**

const { Worker } = require('worker\_threads');

// Inicializa o worker com credenciais e ativos da arbitragem

const worker = new Worker('./workerExecOKXFase2.js --------------------------------------------------------------------------------------------', {

  workerData: {

    okxApiKey: 'ac0bc774-1bad-4da2-83f9-55b8eebb697d',

    okxApiSecret: '4AD9EBBD4A8EEB6526F31B9527545ADC',

    okxPassphrase: 'Aa@066466646',

    ativos: { cripto: 'BTC', dolar: 'USDT', moeda: 'BRL' } // Ativos da arbitragem

  }

});

worker.on('message', (msg) => {

  if (msg.type === 'conexoesProntas') {

    console.log('[mainFase2] Conexões prontas. Enviando ordem...');

    const t0 = Date.now();

    const operacoes = [

      { exchange: 'OKX', symbol: 'BTCUSDT', side: 'buy', type: 'limit', amount: '0.00001', price: '82000', timeInForce: 'ioc' },

      { exchange: 'OKX', symbol: 'BTCBRL', side: 'sell', type: 'market', amount: '0.0009', price: '488000', timeInForce: null },

      { exchange: 'OKX', symbol: 'USDTBRL', side: 'buy', type: 'market', amount: '1.00', price: null, timeInForce: null }

    ];

    const now = new Date();

    const hhmmssSSS = now.toISOString().slice(11, 23).replace(/[:.]/g, '');

    const clientOrderId = `cli${hhmmssSSS}`;

    worker.postMessage({ type: 'executeOrder', order: operacoes[0], clientOrderId, t0 });

  } else if (msg.type === 'orderStatus') {

    console.log(`[mainFase2] Resposta do workerExecOKXFase2: ${JSON.stringify(msg)}`);

  }

});

// Envia a mensagem de inicialização

worker.postMessage({ type: 'init' });

**workerExecOKXFase2.js --------------------------------------------------------------------------------------------**

const { parentPort, workerData } = require('worker\_threads');

const WebSocket = require('ws');

const crypto = require('crypto');

// Cores ANSI para logs

const GREEN = '\x1b[32m';

const BROWN = '\x1b[33m';

const RESET = '\x1b[0m';

// Credenciais da OKX (passadas via workerData pelo main)

const { okxApiKey, okxApiSecret, okxPassphrase } = workerData;

// Mapa para correlacionar IDs internos e da exchange

const orderIdMap = new Map(); // clientOrderId -> { id: id enviado, ordId: id da OKX }

// Configuração modular da OKX

const exchanges = {

  OKX: {

    wsUrl: 'wss://ws.okx.com:8443/ws/v5/private',

    formatOrder: (order) => {

      const now = new Date();

      const hhmmssSSS = now.toISOString().slice(11, 23).replace(/[:.]/g, '');

      let ordType;

      if (order.type === 'limit' && order.timeInForce === 'IOC') {

        ordType = 'ioc';

      } else if (order.type === 'limit' && !order.timeInForce) {

        ordType = 'limit';

      } else if (order.type === 'market' && !order.timeInForce) {

        ordType = 'market';

      } else {

        throw new Error(`Combinação de type (${order.type}) e timeInForce (${order.timeInForce}) não suportada`);

      }

      const orderArgs = {

        instId: order.par,

        side: order.side,

        ordType: ordType,

        sz: order.amount,

        tdMode: 'cash' // Modo spot

      };

      // Ajustes conforme a documentação

      if (ordType === 'market') {

        orderArgs.tgtCcy = 'base\_ccy'; // sz em BTC para ordens a mercado

      } else if (ordType === 'limit' || ordType === 'ioc') {

        orderArgs.px = order.price; // Preço obrigatório para limit/ioc

      }

      return {

        id: `ORD${hhmmssSSS}`,

        op: 'order',

        args: [orderArgs]

      };

    },

    parseStatus: (msg, channel) => {

      if (channel === 'trade') {

        return msg.code === '0' ? 'accepted' : 'rejected';

      } else if (channel === 'orders') {

        return msg.state === 'filled' ? 'filled' : msg.state === 'canceled' ? 'cancelled' : 'live';

      }

    }

  }

};

// Função para formatar objetos nos logs

function formatObject(obj) {

  return JSON.stringify(obj, null, 2)

    .replace(/"([^"]+)": "([^"]+)"/g, `"$1": ${GREEN}"$2"${RESET}`)

    .replace(/"([^"]+)": (\d+(.\d+)?)/g, `"$1": ${BROWN}$2${RESET}`)

    .replace(/"([^"]+)": null/g, `"$1": null`);

}

// Função de log com timestamps absolutos e relativos

let t0; // Armazena o timestamp inicial da arbitragem

function logMessage(channel, message) {

  const abs = Date.now();

  const t = t0 ? abs - t0 : 0; // Calcula tempo transcorrido desde t0

  console.log(`[workerExecOKXFase2 - ${channel}] [abs=${abs} t=${t}] ${message}`);

}

// Gera a assinatura HMAC-SHA256 para autenticação

function generateOKXSignature(timestamp, method, path, body) {

  const prehashString = timestamp + method + path + (body || '');

  return crypto.createHmac('sha256', okxApiSecret).update(prehashString).digest('base64');

}

let ws;

let pingInterval;

// Conecta ao WebSocket da OKX

function connectOKXWebSocket() {

  ws = new WebSocket(exchanges.OKX.wsUrl); // Corrigido o erro de sintaxe

  ws.on('open', () => {

    logMessage('Conexão', 'Conectado ao WebSocket OKX.');

    const timestamp = (Date.now() / 1000).toString();

    const sign = generateOKXSignature(timestamp, 'GET', '/users/self/verify', '');

    ws.send(JSON.stringify({

      op: 'login',

      args: [{

        apiKey: okxApiKey,

        passphrase: okxPassphrase,

        timestamp: timestamp,

        sign: sign

      }]

    }));

  });

  ws.on('message', (data) => {

    if (data.toString() === 'pong') {

      logMessage('Conexão', 'Pong recebido.');

      return;

    }

    let msg;

    try {

      msg = JSON.parse(data);

      logMessage('Geral', `Mensagem recebida:\n${formatObject(msg)}`);

    } catch (e) {

      logMessage('Erro', `Falha ao parsear mensagem: ${data}`);

      return;

    }

    if (msg.event === 'login' && msg.code === '0') {

      logMessage('Conexão', 'Autenticado com sucesso.');

      ws.send(JSON.stringify({

        op: 'subscribe',

        args: [

          { channel: 'orders', instType: 'SPOT', instId: 'BTC-USDT' },

          { channel: 'orders', instType: 'SPOT', instId: 'BTC-BRL' },

          { channel: 'orders', instType: 'SPOT', instId: 'USDT-BRL' }

        ]

      }));

      parentPort.postMessage({ type: 'conexoesProntas' });

      pingInterval = setInterval(() => {

        if (ws.readyState === WebSocket.OPEN) {

          ws.send('ping');

          logMessage('Conexão', 'Ping enviado.');

        }

      }, 25000);

    }

    if (msg.event === 'subscribe' && msg.arg?.channel === 'orders') {

      logMessage('Conexão', `Inscrito no canal orders para ${msg.arg.instId}.`);

    }

    // Resposta do canal de postagem (op: 'order')

    if (msg.op === 'order') {

      const abs = Date.now();

      const t = t0 ? abs - t0 : 0;

      const status = exchanges.OKX.parseStatus(msg, 'trade');

      if (msg.data && msg.data.length > 0) {

        const orderData = msg.data[0];

        const clientOrderId = [...orderIdMap.entries()].find(([\_, ids]) => ids.id === msg.id)?.[0];

        if (clientOrderId && orderData.ordId) {

          orderIdMap.set(clientOrderId, { id: msg.id, ordId: orderData.ordId }); // Atualiza com o ordId da OKX

        }

        logMessage('Postagem', `Resposta de postagem: ${status} para ordem ${orderData.ordId}`);

        parentPort.postMessage({

          type: 'orderStatus',

          channel: 'Postagem',

          orderId: msg.id,

          clientOrderId: clientOrderId || null,

          status: status,

          instId: orderData.instId,

          errorCode: msg.code !== '0' ? msg.code : null,

          errorMsg: msg.code !== '0' ? msg.msg : null,

          abs,

          t

        });

      }

    }

    // Resposta do canal de ordens (orders)

    if (msg.arg?.channel === 'orders' && msg.data?.length > 0) {

      const abs = Date.now();

      const t = t0 ? abs - t0 : 0;

      msg.data.forEach(order => {

        const status = exchanges.OKX.parseStatus(order, 'orders');

        const clientOrderId = [...orderIdMap.entries()].find(([\_, ids]) => ids.ordId === order.ordId)?.[0];

        logMessage('Ordens', `Atualização de ordem: ${status} para ${order.ordId}`);

        parentPort.postMessage({

          type: 'orderStatus',

          channel: 'Ordens',

          orderId: order.ordId,

          clientOrderId: clientOrderId || null,

          status: status,

          instId: order.instId,

          avgPx: order.avgPx || null,

          fillSz: order.fillSz || null,

          cancelSource: order.cancelSource || null,

          abs,

          t

        });

      });

    }

  });

  ws.on('close', () => {

    logMessage('Conexão', 'WebSocket desconectado. Tentando reconectar em 5s...');

    clearInterval(pingInterval);

    setTimeout(connectOKXWebSocket, 5000);

  });

  ws.on('error', (err) => {

    logMessage('Erro', `Erro no WebSocket: ${err.message}`);

  });

}

// Envia a ordem formatada para a OKX

function sendOrder(order, clientOrderId, receivedT0) {

  const formattedOrder = exchanges.OKX.formatOrder(order);

  orderIdMap.set(clientOrderId, { id: formattedOrder.id, ordId: null }); // Inicializa com id enviado

  logMessage('Postagem', `Enviando ordem (clientOrderId: ${clientOrderId}):\n${formatObject(formattedOrder)}`);

  ws.send(JSON.stringify(formattedOrder));

}

// Inicializa a conexão e escuta mensagens do mainFase2

parentPort.on('message', (msg) => {

  if (msg.type === 'init') {

    logMessage('Geral', 'Inicializando workerExecOKXFase2.');

    connectOKXWebSocket();

  } else if (msg.type === 'executeOrder') {

    t0 = msg.t0; // Armazena o t0 recebido do mainFase2

    logMessage('Geral', `Recebida ordem do mainFase2 (clientOrderId: ${msg.clientOrderId}):\n${formatObject(msg.order)}`);

    sendOrder(msg.order, msg.clientOrderId, msg.t0);

  }

});

**ATUALIZAÇÃO DO DESENVOLVIMENTO NO 07/04/2025.**

Passamos ao desenvolvimento da Fase 3. Foram criados dois códigos, o coordenador.js (que por enquanto não é worker de ninguém) e o workerExecOKX.js, que é um worker do coordenador. Na sequencia se apresentam os códigos atualizados.

Como fora definido na doc e na na logica das arbitragens, o coordenador controla a arbitragem e envia as ordens ao worker correspondente. Até agora so existe um worker, o da OKX mais na próxima etapa sera desenvolvido o worker da Bybit: workerExecBybit.js

Foi ajustada também a nomenclatura da matriz de operações e o mapeamento dos campos para as diferente exchanges.

**ATUALIZAÇÃO DO DESENVOLVIMENTO NO 08/04/2025.**

Iniciou a programação do worker da Bybit, workerExecBybit.js. Houve uma confusão importante na dinâmica da arbitragem e das mensagens dos trades (T1, T2 e T3). No fim foi resolvido segundo a explicação da sequencia. Os códigos finais do coordenador e do worker se encontram após a explicação.

**Parte 1: Sequência Lógica das Mensagens**

**Contexto Geral**

O sistema envolve o coordenador.js gerenciando a arbitragem, enviando ordens T1, T2 e T3 para o workerExecBybit.js, que utiliza dois WebSockets: tradeWs para postar ordens (canal "Trade") e orderWs para receber atualizações de status (canal "Ordens"). O objetivo é enviar T1 primeiro, disparando T2 e T3 somente se T1 for preenchida (filled). O processo pode abortar por rejeição, cancelamento ou timeout. Para exemplo, consideramos uma matriz com T1 como uma ordem limite IOC para BTCUSDT e T2 como uma ordem de mercado para USDTBRL.

**Sequência Detalhada**

Assumirei que o sistema já está autenticado, com tradeWs e orderWs conectados e inscritos no tópico "order". Vou descrever cada etapa e cobrir todos os desfechos possíveis.

**1. Envio da Ordem T1 (Postagem Inicial no tradeWs)**

O processo começa quando o coordenador.js inicia a arbitragem chamando a função startArbitrage(), que envia T1 ao workerExecBybit.js. O coordenador registra que está enviando a ordem para a Bybit, indicando o par BTCUSDT como uma ordem limite de venda com quantidade 0.0001, preço 76500 e timeInForce IOC. No workerExecBybit.js, a mensagem é recebida via parentPort. Um clientOrderId único é gerado, como CLI214432676, e a ordem é formatada para a API da Bybit, criando um formattedOrder.id, como ORD214432678. O orderIdMap é inicializado com a associação entre o clientOrderId e o formattedOrder.id. A ordem é então enviada ao tradeWs, registrando que está postando a ordem com o clientOrderId especificado.

**2. Resposta do tradeWs (Canal "Trade" - accepted ou rejected)**

Se a Bybit aceita a ordem (caso de sucesso), o tradeWs retorna uma mensagem confirmando que a ordem foi criada com sucesso, fornecendo um orderId oficial da Bybit, como 1924452983358359808. No workerExecBybit.js, o clientOrderId é buscado no orderIdMap usando o formattedOrder.id enviado (ORD214432678), encontrando CLI214432676. O orderIdMap é atualizado para associar CLI214432676 ao orderId da Bybit (1924452983358359808). O status accepted é enviado ao coordenador.js, que registra que T1 foi aceita e está aguardando o status final. Se a Bybit rejeitar a ordem (caso de falha), como por quantidade inválida, o tradeWs retorna uma mensagem de erro, como "Order quantity exceeded lower limit". O workerExecBybit.js envia o status rejected ao coordenador.js com o código de erro e a mensagem correspondente. O coordenador.js registra que T1 foi rejeitada, incluindo o motivo do erro, e aborta o processo, encerrando sem enviar T2 ou T3.

**3. Atualização no orderWs (Canal "Ordens" - filled, cancelled, ou Timeout)**

No caso de sucesso (filled), o orderWs atualiza o status da ordem, indicando que o orderId 1924452983358359808 foi preenchido, com preço médio 77029.8 e quantidade executada 0.0001. O workerExecBybit.js verifica o orderIdMap, encontra CLI214432676 associado ao orderId, e registra que a ordem foi preenchida. O status filled é enviado ao coordenador.js, que marca t1Filled como true, limpa o timeout e registra que T1 foi preenchida, disparando T2 e T3. No caso de cancelamento (cancelled), se a ordem IOC não encontrar liquidez imediata, o orderWs retorna que o orderId foi cancelado, com um motivo como "EC\_NoImmediateFill". O workerExecBybit.js busca o clientOrderId no orderIdMap, registra o cancelamento e envia o status cancelled ao coordenador.js. O coordenador registra que T1 foi cancelada, indicando o motivo, e aborta o processo, não enviando T2 ou T3. Se nenhuma mensagem filled ou cancelled chegar dentro de 3 segundos (timeout), o coordenador.js registra que o tempo limite de 3s para T1 foi atingido e aborta, definindo aborted como true, encerrando sem enviar T2 ou T3.

**4. Envio de T2/T3 (Postagem no tradeWs)**

O envio de T2 e T3 só ocorre se T1 for preenchida. O coordenador.js registra que está enviando T2 para a Bybit, como uma ordem de mercado de venda para USDTBRL com quantidade 1. No workerExecBybit.js, um novo clientOrderId é gerado, como CLI214433039, e a ordem é formatada com um formattedOrder.id, como ORD214433040. O orderIdMap é atualizado, adicionando a nova associação, mantendo a entrada de T1 intacta. A ordem é enviada ao tradeWs, registrando a postagem com o clientOrderId correspondente.

**5. Resposta do tradeWs para T2/T3**

Em caso de sucesso (accepted), o tradeWs retorna que a ordem foi aceita, fornecendo um novo orderId, como 1924452986327927040. O workerExecBybit.js atualiza o orderIdMap, associando CLI214433039 ao novo orderId, e envia o status accepted ao coordenador.js, que registra a aceitação. Se houver rejeição, como por quantidade inválida, o tradeWs retorna um erro, como "Order quantity exceeded lower limit". O workerExecBybit.js envia o status rejected ao coordenador.js, que registra a rejeição, mas o processo continua, pois T2/T3 não têm dependências adicionais.

**6. Atualização no orderWs para T2/T3**

No caso de sucesso (filled), o orderWs retorna que o orderId 1924452986327927040 foi preenchido, com preço médio 6.025 e quantidade 1. O workerExecBybit.js busca CLI214433039 no orderIdMap, registra o preenchimento e envia o status filled ao coordenador.js, que registra que T2/T3 foi preenchida, concluindo a arbitragem com sucesso. Em caso de cancelamento (cancelled), raro para ordens de mercado, o orderWs indicaria que a ordem foi cancelada. O workerExecBybit.js enviaria o status cancelled ao coordenador.js, que registraria o evento, encerrando o processo com T2/T3 não preenchidos.

**Parte 2: Relação entre orderIdMap, clientOrderId, orderId e a Confusão**

**Definições**

O clientOrderId é um identificador gerado pelo coordenador.js, como CLI214432676, para rastrear a ordem internamente, único por operação. O formattedOrder.id é um ID temporário criado no workerExecBybit.js, como ORD214432678, ao formatar a ordem para envio ao tradeWs. O orderId é o identificador oficial retornado pela Bybit, como 1924452983358359808, após aceitar a ordem. O orderIdMap é um mapa no workerExecBybit.js que correlaciona clientOrderId ao orderId (ou inicialmente ao formattedOrder.id).

**Dinâmica e Confusão**

Na postagem inicial de T1, o coordenador.js envia a ordem com clientOrderId CLI214432676. O workerExecBybit.js cria formattedOrder.id ORD214432678 e adiciona ao orderIdMap a associação CLI214432676 com ORD214432678, enviando a ordem ao tradeWs. Quando o tradeWs responde, a Bybit retorna o orderId 1924452983358359808. No código original com bug, o workerExecBybit.js buscava o clientOrderId usando msg.id, que não existia na resposta, resultando em null. Isso criava uma entrada inválida no orderIdMap, mantendo CLI214432676 associado a ORD214432678 e adicionando uma nova entrada null com 1924452983358359808. Após a correção, o código busca o clientOrderId pelo formattedOrder.id (ORD214432678) no orderIdMap, atualizando a associação para CLI214432676 com 1924452983358359808.

Quando o orderWs atualiza o status com orderId 1924452983358359808 como filled, o workerExecBybit.js busca no orderIdMap e encontra CLI214432676, enviando o status filled ao coordenador.js. No bug original, a busca falhava porque 1924452983358359808 estava associado a null, não a CLI214432676, impedindo o envio do filled e o disparo de T2/T3. Com a correção, o filled é enviado corretamente, permitindo o envio de T2/T3. Para T2, o processo se repete: CLI214433039 é associado a ORD214433040, enviado ao tradeWs, atualizado para 1924452986327927040 no orderIdMap, e o status filled é processado sem problemas.

A confusão surgiu porque a Bybit não retorna o formattedOrder.id (ORD...) na resposta do tradeWs, apenas o orderId. O código original assumia que msg.id seria o ORD..., mas como isso não acontecia, o mapeamento quebrava. A solução foi usar o ORD... já presente no orderIdMap como chave temporária para encontrar o clientOrderId correto.

**Resumo para Documentação**

O fluxo segue do coordenador.js para o tradeWs (postagem), recebendo accepted ou rejected, seguido pelo orderWs (status) com filled, cancelled ou timeout, e finalmente T2/T3 são enviados se T1 for preenchida. O mapeamento envolve clientOrderId (interno) para formattedOrder.id (envio) e depois orderId (Bybit), gerenciado pelo orderIdMap. A lição para outras exchanges é garantir que o ID enviado seja rastreável na resposta ou usar um mapa robusto para correlacionar IDs internos e externos.

**Proxima Tarefa**

Continuar testando o coordenador e seu funcionamento com as duas exchanges programadas, Bybit e OKX.

**Códigos no Estado Atual**

**//#####################################################**

**workerExecBybit.js --------------------------------------------------------------------------------------------**

const { parentPort, workerData } = require('worker\_threads');

const WebSocket = require('ws');

const crypto = require('crypto');

// Cores ANSI para logs

const GREEN = '\x1b[32m';

const BROWN = '\x1b[33m';

const RED = '\x1b[31m';

const RESET = '\x1b[0m';

// Credenciais da Bybit

const { bybitApiKey, bybitApiSecret } = workerData;

// Mapa para correlacionar IDs

const orderIdMap = new Map();

// Contador para numerar mensagens do WebSocket de ordens

let orderMessageCounter = 0;

// Configuração da Bybit

const exchanges = {

  Bybit: {

    tradeWsUrl: 'wss://stream.bybit.com/v5/trade',

    orderWsUrl: 'wss://stream.bybit.com/v5/private',

    formatOrder: (order) => {

      const now = new Date();

      const hhmmssSSS = now.toISOString().slice(11, 23).replace(/[:.]/g, '');

      const timestamp = Date.now().toString();

      let orderType;

      if (order.type === 'limit' && order.timeInForce?.toLowerCase() === 'ioc') {

        orderType = 'Limit';

      } else if (order.type === 'limit' && !order.timeInForce) {

        orderType = 'Limit';

      } else if (order.type === 'market' && !order.timeInForce) {

        orderType = 'Market';

      } else {

        throw new Error(`Tipo inválido: ${order.type}/${order.timeInForce}`);

      }

      const params = {

        category: 'spot',

        symbol: order.symbol,

        side: order.side.charAt(0).toUpperCase() + order.side.slice(1).toLowerCase(),

        orderType,

        isLeverage: '1',

        qty: order.amount,

        timestamp,

      };

      if (orderType === 'Limit') {

        params.price = order.price;

        if (order.timeInForce?.toLowerCase() === 'ioc') {

          params.timeInForce = 'IOC';

        }

      }

      return {

        id: `ORD${hhmmssSSS}`,

        header: {

          "X-BAPI-TIMESTAMP": timestamp,

          "X-BAPI-RECV-WINDOW": "2000"

        },

        op: 'order.create',

        args: [params],

      };

    },

    parseStatus: (msg, channel) => {

      if (channel === 'trade') {

        return msg.retCode === 0 ? 'accepted' : 'rejected';

      } else if (channel === 'orders') {

        return msg.orderStatus === 'Filled' ? 'filled' :

               msg.orderStatus === 'Cancelled' ? 'cancelled' :

               msg.orderStatus === 'New' ? 'live' : 'unknown';

      }

    },

  },

};

// Função para formatar objetos nos logs

function formatObject(obj) {

  return JSON.stringify(obj, null, 2)

    .replace(/"([^"]+)": "([^"]+)"/g, `"$1": ${GREEN}"$2"${RESET}`)

    .replace(/"([^"]+)": (\d+(.\d+)?)/g, `"$1": ${BROWN}$2${RESET}`)

    .replace(/"([^"]+)": null/g, `"$1": null`);

}

// Função de log com timestamps

let t0;

function logMessage(channel, message) {

  const abs = Date.now();

  const t = t0 ? abs - t0 : 0;

  console.log(`[workerExecBybit - ${channel}] [abs=${abs} t=${t}] ${message}`);

}

// Gera assinatura HMAC-SHA256 para autenticação do WebSocket

function generateBybitSignature(expires) {

  const prehashString = `GET/realtime${expires}`;

  return crypto.createHmac('sha256', bybitApiSecret).update(prehashString).digest('hex');

}

let tradeWs;

let orderWs;

let pingIntervalTrade;

let pingIntervalOrder;

let portToCoord;

function connectTradeWebSocket() {

  tradeWs = new WebSocket(exchanges.Bybit.tradeWsUrl);

  tradeWs.on('open', () => {

    logMessage('Trade', 'Conectado ao WebSocket de trades Bybit.');

    const expires = (Date.now() + 10000).toString();

    const signature = generateBybitSignature(expires);

    tradeWs.send(JSON.stringify({

      op: 'auth',

      args: [bybitApiKey, expires, signature],

    }));

  });

  tradeWs.on('message', (data) => {

    let msg;

    try {

      msg = JSON.parse(data);

      logMessage('Trade', `Mensagem recebida:\n${formatObject(msg)}`);

    } catch (e) {

      logMessage('Erro', `Erro ao parsear: ${data}`);

      return;

    }

    if (msg.op === 'auth' && msg.retCode === 0) {

      logMessage('Trade', 'Autenticado com sucesso no WebSocket de trades.');

    }

    if (msg.op === 'order.create') {

      const abs = Date.now();

      const t = t0 ? abs - t0 : 0;

      const status = exchanges.Bybit.parseStatus(msg, 'trade');

      // Busca o clientOrderId pelo formattedOrder.id enviado

      const sentOrder = [...orderIdMap.entries()].find(([\_, id]) => id.startsWith('ORD'));

      const clientOrderId = sentOrder ? sentOrder[0] : null;

      if (msg.retCode === 0 && msg.data?.orderId) {

        if (clientOrderId) {

          orderIdMap.set(clientOrderId, msg.data.orderId); // Atualiza para orderId da Bybit

          logMessage('Postagem', `Postagem: ${status} para ${msg.data.orderId} (clientOrderId: ${clientOrderId})`);

          portToCoord.postMessage({

            type: 'orderStatus',

            channel: 'Postagem',

            orderId: clientOrderId,

            status,

            instId: msg.data.symbol,

            abs,

            t,

          });

        } else {

          logMessage('Postagem', `Erro: Nenhum clientOrderId encontrado para atualizar com orderId: ${msg.data.orderId}`);

        }

      } else {

        logMessage('Postagem', `Postagem rejeitada: ${msg.retMsg}`);

        portToCoord.postMessage({

          type: 'orderStatus',

          channel: 'Postagem',

          orderId: clientOrderId || 'unknown',

          status,

          instId: msg.args?.[0]?.symbol || 'unknown',

          errorCode: msg.retCode,

          errorMsg: msg.retMsg,

          abs,

          t,

        });

      }

    }

  });

  tradeWs.on('close', () => {

    logMessage('Trade', 'Trade WebSocket desconectado. Reconectando em 5s...');

    clearInterval(pingIntervalTrade);

    setTimeout(connectTradeWebSocket, 5000);

  });

  tradeWs.on('error', (err) => {

    logMessage('Erro', `Erro no Trade WebSocket: ${err.message}`);

  });

}

function connectOrderWebSocket() {

  orderWs = new WebSocket(exchanges.Bybit.orderWsUrl);

  orderWs.on('open', () => {

    logMessage('Orders', 'Conectado ao WebSocket de ordens Bybit.');

    const expires = (Date.now() + 10000).toString();

    const signature = generateBybitSignature(expires);

    orderWs.send(JSON.stringify({

      op: 'auth',

      args: [bybitApiKey, expires, signature],

    }));

  });

  orderWs.on('message', (data) => {

    orderMessageCounter++;

    let msg;

    try {

      msg = JSON.parse(data);

      logMessage('Orders', `${RED}========== Mensagem WS Orders [${orderMessageCounter}] ==========${RESET}\n${formatObject(msg)}`);

      logMessage('Orders', `Estado do orderIdMap: ${JSON.stringify([...orderIdMap.entries()])}`);

    } catch (e) {

      logMessage('Erro', `Erro ao parsear: ${data}`);

      return;

    }

    if (msg.op === 'auth' && msg.success) {

      logMessage('Orders', 'Autenticado com sucesso no WebSocket de ordens.');

      orderWs.send(JSON.stringify({

        op: 'subscribe',

        args: ['order'],

      }));

    }

    if (msg.op === 'subscribe' && msg.success) {

      logMessage('Orders', 'Inscrito no tópico de ordens.');

      portToCoord.postMessage({ type: 'conexoesProntas' });

    }

    if (msg.topic === 'order' && msg.data?.length > 0) {

      const abs = Date.now();

      const t = t0 ? abs - t0 : 0;

      msg.data.forEach((order) => {

        const status = exchanges.Bybit.parseStatus(order, 'orders');

        const clientOrderId = [...orderIdMap.entries()].find(([\_, id]) => id === order.orderId)?.[0];

        if (!clientOrderId) {

          logMessage('Orders', `Nenhum clientOrderId encontrado para orderId: ${order.orderId}`);

          return;

        }

        logMessage('Ordens', `Ordem: ${status} para ${order.orderId} (clientOrderId: ${clientOrderId})`);

        portToCoord.postMessage({

          type: 'orderStatus',

          channel: 'Ordens',

          orderId: clientOrderId,

          status,

          instId: order.symbol,

          avgPx: order.avgPrice || null,

          fillSz: order.cumExecQty || null,

          cancelSource: order.rejectReason || null,

          abs,

          t,

        });

      });

    }

  });

  orderWs.on('close', () => {

    logMessage('Orders', 'Order WebSocket desconectado. Reconectando em 5s...');

    clearInterval(pingIntervalOrder);

    setTimeout(connectOrderWebSocket, 5000);

  });

  orderWs.on('error', (err) => {

    logMessage('Erro', `Erro no Order WebSocket: ${err.message}`);

  });

}

function sendOrder(order, clientOrderId) {

  const formattedOrder = exchanges.Bybit.formatOrder(order);

  orderIdMap.set(clientOrderId, formattedOrder.id); // Map inicial

  logMessage('Postagem', `Enviando (clientOrderId: ${clientOrderId}):\n${formatObject(formattedOrder)}`);

  tradeWs.send(JSON.stringify(formattedOrder));

}

parentPort.on('message', (msg) => {

  if (msg.type === 'init') {

    portToCoord = msg.port;

    logMessage('Geral', 'Inicializando workerExecBybit.');

    connectTradeWebSocket();

    connectOrderWebSocket();

    pingIntervalTrade = setInterval(() => {

      if (tradeWs && tradeWs.readyState === WebSocket.OPEN) {

        tradeWs.send(JSON.stringify({ op: 'ping' }));

        logMessage('Trade', 'Ping enviado.');

      }

    }, 20000);

    pingIntervalOrder = setInterval(() => {

      if (orderWs && orderWs.readyState === WebSocket.OPEN) {

        orderWs.send(JSON.stringify({ op: 'ping' }));

        logMessage('Orders', 'Ping enviado.');

      }

    }, 20000);

  } else if (msg.type === 'executeOrder') {

    t0 = t0 || Date.now();

    logMessage('Geral', `Recebida ordem (clientOrderId: ${msg.orderId}):\n${formatObject(msg.order)}`);

    sendOrder(msg.order, msg.orderId);

  }

});

**//#####################################################**

workerExecOKX.js

"use strict";

const { parentPort, workerData } = require('worker\_threads');

const WebSocket = require('ws');

const crypto = require('crypto');

// Cores ANSI para logs

const GREEN = '\x1b[32m';

const BROWN = '\x1b[33m';

const RESET = '\x1b[0m';

// Credenciais da OKX

const { okxApiKey, okxApiSecret, okxPassphrase } = workerData;

// Validação das credenciais

if (!okxApiKey || !okxApiSecret || !okxPassphrase) {

  console.error('[workerExecOKX] Erro: Credenciais incompletas no workerData:', workerData);

  throw new Error('Credenciais da OKX não fornecidas corretamente');

}

// Mapa para correlacionar IDs

const orderIdMap = new Map();

// Configuração da OKX

const exchanges = {

  OKX: {

    wsUrl: 'wss://ws.okx.com:8443/ws/v5/private',

    formatOrder: (order) => {

      const now = new Date();

      const hhmmssSSS = now.toISOString().slice(11, 23).replace(/[:.]/g, '');

      let ordType;

      if (order.type === 'limit' && order.timeInForce?.toLowerCase() === 'ioc') {

        ordType = 'ioc';

      } else if (order.type === 'limit' && !order.timeInForce) {

        ordType = 'limit';

      } else if (order.type === 'market' && !order.timeInForce) {

        ordType = 'market';

      } else {

        throw new Error(`Tipo inválido: ${order.type}/${order.timeInForce}`);

      }

      const orderArgs = {

        instId: order.symbol.replace(/(.+)(USDT|BRL)/, '$1-$2'),

        side: order.side,

        ordType,

        sz: order.amount,

        tdMode: 'cash'

      };

      if (ordType === 'market') orderArgs.tgtCcy = 'base\_ccy';

      else if (ordType === 'limit' || ordType === 'ioc') orderArgs.px = order.price;

      return {

        id: `ORD${hhmmssSSS}`,

        op: 'order',

        args: [orderArgs]

      };

    },

    parseStatus: (msg, channel) => {

      if (channel === 'trade') return msg.code === '0' ? 'accepted' : 'rejected';

      else if (channel === 'orders') {

        return msg.state === 'filled' ? 'filled' : msg.state === 'canceled' ? 'cancelled' : 'live';

      }

    }

  }

};

// Função para formatar objetos

function formatObject(obj) {

  return JSON.stringify(obj, null, 2)

    .replace(/"([^"]+)": "([^"]+)"/g, `"$1": ${GREEN}"$2"${RESET}`)

    .replace(/"([^"]+)": (\d+(.\d+)?)/g, `"$1": ${BROWN}$2${RESET}`)

    .replace(/"([^"]+)": null/g, `"$1": null`);

}

// Função de log com timestamps

let t0;

function logMessage(channel, message) {

  const abs = Date.now();

  const t = t0 ? abs - t0 : 0;

  console.log(`[workerExecOKX - ${channel}] [abs=${abs} t=${t}] ${message}`);

}

// Gera assinatura para autenticação

function generateOKXSignature(timestamp, method, path, body) {

  const prehashString = timestamp + method + path + (body || '');

  return crypto.createHmac('sha256', okxApiSecret).update(prehashString).digest('base64');

}

let ws;

let pingInterval;

let portToCoord;

// Conecta ao WebSocket da OKX

function connectOKXWebSocket() {

  ws = new WebSocket(exchanges.OKX.wsUrl);

  ws.on('open', () => {

    const timestamp = (Date.now() / 1000).toString();

    const sign = generateOKXSignature(timestamp, 'GET', '/users/self/verify', '');

    ws.send(JSON.stringify({

      op: 'login',

      args: [{ apiKey: okxApiKey, passphrase: okxPassphrase, timestamp, sign }]

    }));

  });

  ws.on('message', (data) => {

    if (data.toString() === 'pong') {

      return;

    }

    let msg;

    try {

      msg = JSON.parse(data);

    } catch (e) {

      logMessage('Erro', `Erro ao parsear: ${data}`);

      return;

    }

    if (msg.event === 'login' && msg.code === '0') {

      ws.send(JSON.stringify({

        op: 'subscribe',

        args: [

          { channel: 'orders', instType: 'SPOT', instId: 'BTC-USDT' },

          { channel: 'orders', instType: 'SPOT', instId: 'BTC-BRL' },

          { channel: 'orders', instType: 'SPOT', instId: 'USDT-BRL' }

        ]

      }));

    }

    if (msg.event === 'subscribe' && msg.arg?.channel === 'orders') {

      if (msg.arg.instId === 'USDT-BRL') {

        portToCoord.postMessage({ type: 'conexoesProntas' });

      }

    }

    if (msg.op === 'order') {

      const abs = Date.now();

      const t = t0 ? abs - t0 : 0;

      const status = exchanges.OKX.parseStatus(msg, 'trade');

      if (msg.data && msg.data.length > 0) {

        const orderData = msg.data[0];

        const clientOrderId = [...orderIdMap.entries()].find(([\_, ids]) => ids.id === msg.id)?.[0];

        if (clientOrderId && orderData.ordId) {

          orderIdMap.set(clientOrderId, { id: msg.id, ordId: orderData.ordId });

        }

        logMessage('Postagem', `Postagem: ${status} para ${orderData.instId} - Motivo: ${orderData.sMsg || 'Desconhecido'}`);

        portToCoord.postMessage({

          type: 'orderStatus',

          channel: 'Postagem',

          orderId: clientOrderId,

          status,

          instId: orderData.instId,

          errorCode: msg.code !== '0' ? msg.code : null,

          errorMsg: msg.code !== '0' ? msg.msg : null,

          abs,

          t

        });

      }

    }

    if (msg.arg?.channel === 'orders' && msg.data?.length > 0) {

      const abs = Date.now();

      const t = t0 ? abs - t0 : 0;

      msg.data.forEach(order => {

        const status = exchanges.OKX.parseStatus(order, 'orders');

        const clientOrderId = [...orderIdMap.entries()].find(([\_, ids]) => ids.ordId === order.ordId)?.[0];

        logMessage('Ordens', `Ordem: ${status} para ${order.ordId}`);

        portToCoord.postMessage({

          type: 'orderStatus',

          channel: 'Ordens',

          orderId: clientOrderId,

          status,

          instId: order.instId,

          avgPx: order.avgPx || null,

          fillSz: order.fillSz || null,

          cancelSource: order.cancelSource || null,

          abs,

          t

        });

      });

    }

  });

  ws.on('close', () => {

    clearInterval(pingInterval);

    setTimeout(connectOKXWebSocket, 5000);

  });

  ws.on('error', (err) => {

    logMessage('Erro', `Erro no WebSocket: ${err.message}`);

  });

}

// Envia ordem à OKX

function sendOrder(order, clientOrderId) {

  const formattedOrder = exchanges.OKX.formatOrder(order);

  orderIdMap.set(clientOrderId, { id: formattedOrder.id, ordId: null });

  logMessage('Postagem', `Enviando (clientOrderId: ${clientOrderId}):\n${formatObject(formattedOrder)}`);

  ws.send(JSON.stringify(formattedOrder));

}

// Inicialização

parentPort.on('message', (msg) => {

  if (msg.type === 'init') {

    portToCoord = msg.port;

    logMessage('Geral', 'Inicializando workerExecOKX.');

    connectOKXWebSocket();

    pingInterval = setInterval(() => {

      if (ws && ws.readyState === WebSocket.OPEN) {

        ws.send('ping');

      }

    }, 25000);

  } else if (msg.type === 'executeOrder') {

    t0 = t0 || Date.now();

    logMessage('Geral', `Recebida ordem (clientOrderId: ${msg.orderId}):\n${formatObject(msg.order)}`);

    sendOrder(msg.order, msg.orderId);

  }

});

**//#####################################################**

Worker coordenador.js

"use strict";

const { parentPort, workerData } = require('worker\_threads');

const WebSocket = require('ws');

const crypto = require('crypto');

// Cores ANSI para logs

const GREEN = '\x1b[32m';

const BROWN = '\x1b[33m';

const RESET = '\x1b[0m';

// Credenciais da OKX

const { okxApiKey, okxApiSecret, okxPassphrase } = workerData;

// Validação das credenciais

if (!okxApiKey || !okxApiSecret || !okxPassphrase) {

  console.error('[workerExecOKX] Erro: Credenciais incompletas no workerData:', workerData);

  throw new Error('Credenciais da OKX não fornecidas corretamente');

}

// Mapa para correlacionar IDs

const orderIdMap = new Map();

// Configuração da OKX

const exchanges = {

  OKX: {

    wsUrl: 'wss://ws.okx.com:8443/ws/v5/private',

    formatOrder: (order) => {

      const now = new Date();

      const hhmmssSSS = now.toISOString().slice(11, 23).replace(/[:.]/g, '');

      let ordType;

      if (order.type === 'limit' && order.timeInForce?.toLowerCase() === 'ioc') {

        ordType = 'ioc';

      } else if (order.type === 'limit' && !order.timeInForce) {

        ordType = 'limit';

      } else if (order.type === 'market' && !order.timeInForce) {

        ordType = 'market';

      } else {

        throw new Error(`Tipo inválido: ${order.type}/${order.timeInForce}`);

      }

      const orderArgs = {

        instId: order.symbol.replace(/(.+)(USDT|BRL)/, '$1-$2'),

        side: order.side,

        ordType,

        sz: order.amount,

        tdMode: 'cash'

      };

      if (ordType === 'market') orderArgs.tgtCcy = 'base\_ccy';

      else if (ordType === 'limit' || ordType === 'ioc') orderArgs.px = order.price;

      return {

        id: `ORD${hhmmssSSS}`,

        op: 'order',

        args: [orderArgs]

      };

    },

    parseStatus: (msg, channel) => {

      if (channel === 'trade') return msg.code === '0' ? 'accepted' : 'rejected';

      else if (channel === 'orders') {

        return msg.state === 'filled' ? 'filled' : msg.state === 'canceled' ? 'cancelled' : 'live';

      }

    }

  }

};

// Função para formatar objetos

function formatObject(obj) {

  return JSON.stringify(obj, null, 2)

    .replace(/"([^"]+)": "([^"]+)"/g, `"$1": ${GREEN}"$2"${RESET}`)

    .replace(/"([^"]+)": (\d+(.\d+)?)/g, `"$1": ${BROWN}$2${RESET}`)

    .replace(/"([^"]+)": null/g, `"$1": null`);

}

// Função de log com timestamps

let t0;

function logMessage(channel, message) {

  const abs = Date.now();

  const t = t0 ? abs - t0 : 0;

  console.log(`[workerExecOKX - ${channel}] [abs=${abs} t=${t}] ${message}`);

}

// Gera assinatura para autenticação

function generateOKXSignature(timestamp, method, path, body) {

  const prehashString = timestamp + method + path + (body || '');

  return crypto.createHmac('sha256', okxApiSecret).update(prehashString).digest('base64');

}

let ws;

let pingInterval;

let portToCoord;

// Conecta ao WebSocket da OKX

function connectOKXWebSocket() {

  ws = new WebSocket(exchanges.OKX.wsUrl);

  ws.on('open', () => {

    const timestamp = (Date.now() / 1000).toString();

    const sign = generateOKXSignature(timestamp, 'GET', '/users/self/verify', '');

    ws.send(JSON.stringify({

      op: 'login',

      args: [{ apiKey: okxApiKey, passphrase: okxPassphrase, timestamp, sign }]

    }));

  });

  ws.on('message', (data) => {

    if (data.toString() === 'pong') {

      return;

    }

    let msg;

    try {

      msg = JSON.parse(data);

    } catch (e) {

      logMessage('Erro', `Erro ao parsear: ${data}`);

      return;

    }

    if (msg.event === 'login' && msg.code === '0') {

      ws.send(JSON.stringify({

        op: 'subscribe',

        args: [

          { channel: 'orders', instType: 'SPOT', instId: 'BTC-USDT' },

          { channel: 'orders', instType: 'SPOT', instId: 'BTC-BRL' },

          { channel: 'orders', instType: 'SPOT', instId: 'USDT-BRL' }

        ]

      }));

    }

    if (msg.event === 'subscribe' && msg.arg?.channel === 'orders') {

      if (msg.arg.instId === 'USDT-BRL') {

        portToCoord.postMessage({ type: 'conexoesProntas' });

      }

    }

    if (msg.op === 'order') {

      const abs = Date.now();

      const t = t0 ? abs - t0 : 0;

      const status = exchanges.OKX.parseStatus(msg, 'trade');

      if (msg.data && msg.data.length > 0) {

        const orderData = msg.data[0];

        const clientOrderId = [...orderIdMap.entries()].find(([\_, ids]) => ids.id === msg.id)?.[0];

        if (clientOrderId && orderData.ordId) {

          orderIdMap.set(clientOrderId, { id: msg.id, ordId: orderData.ordId });

        }

        logMessage('Postagem', `Postagem: ${status} para ${orderData.instId} - Motivo: ${orderData.sMsg || 'Desconhecido'}`);

        portToCoord.postMessage({

          type: 'orderStatus',

          channel: 'Postagem',

          orderId: clientOrderId,

          status,

          instId: orderData.instId,

          errorCode: msg.code !== '0' ? msg.code : null,

          errorMsg: msg.code !== '0' ? msg.msg : null,

          abs,

          t

        });

      }

    }

    if (msg.arg?.channel === 'orders' && msg.data?.length > 0) {

      const abs = Date.now();

      const t = t0 ? abs - t0 : 0;

      msg.data.forEach(order => {

        const status = exchanges.OKX.parseStatus(order, 'orders');

        const clientOrderId = [...orderIdMap.entries()].find(([\_, ids]) => ids.ordId === order.ordId)?.[0];

        logMessage('Ordens', `Ordem: ${status} para ${order.ordId}`);

        portToCoord.postMessage({

          type: 'orderStatus',

          channel: 'Ordens',

          orderId: clientOrderId,

          status,

          instId: order.instId,

          avgPx: order.avgPx || null,

          fillSz: order.fillSz || null,

          cancelSource: order.cancelSource || null,

          abs,

          t

        });

      });

    }

  });

  ws.on('close', () => {

    clearInterval(pingInterval);

    setTimeout(connectOKXWebSocket, 5000);

  });

  ws.on('error', (err) => {

    logMessage('Erro', `Erro no WebSocket: ${err.message}`);

  });

}

// Envia ordem à OKX

function sendOrder(order, clientOrderId) {

  const formattedOrder = exchanges.OKX.formatOrder(order);

  orderIdMap.set(clientOrderId, { id: formattedOrder.id, ordId: null });

  logMessage('Postagem', `Enviando (clientOrderId: ${clientOrderId}):\n${formatObject(formattedOrder)}`);

  ws.send(JSON.stringify(formattedOrder));

}

// Inicialização

parentPort.on('message', (msg) => {

  if (msg.type === 'init') {

    portToCoord = msg.port;

    logMessage('Geral', 'Inicializando workerExecOKX.');

    connectOKXWebSocket();

    pingInterval = setInterval(() => {

      if (ws && ws.readyState === WebSocket.OPEN) {

        ws.send('ping');

      }

    }, 25000);

  } else if (msg.type === 'executeOrder') {

    t0 = t0 || Date.now();

    logMessage('Geral', `Recebida ordem (clientOrderId: ${msg.orderId}):\n${formatObject(msg.order)}`);

    sendOrder(msg.order, msg.orderId);

  }

});

**Proxima Etapa de Desenvolvimento (17/04/2025).**

Continuar o desenvolvimento do worker da Binance, que já esta na fase 3 mas ainda esta incompleto. O worker é ativado pelo coordenador mas so posta uma ordem fixa. Ou seja, ele ainda nao processa ordens da matriz de operações. Apenas estamos desenvolvendo o dialogo worker – plataforma e vice-versa, para depois sim, incorporar a integração ao coordenador com o processamento correto das operações da matriz e da dinâmica da arbitragem.

Seguem dois códigos. Um é do worker no estado atual e o outro é de um programa simples que posta uma operação. Este ultimo, em outros chats, esta sendo utilizado como modelo para o desenvolvimento.

Este worker nao necessita do seguimento das ordens através do orderId, clientorderid, etc., porque o resultado da postagem e os detalhes da ordem chegam pelo mesmo canal que a ordem é enviada (diferentemente da OKX e Bybit).

**//#####################################################**

workerExecBinance.js

"use strict";

const { parentPort, workerData } = require('worker\_threads');

const WebSocket = require('ws');

const crypto = require('crypto');

// Cores ANSI para logs

const GREEN = '\x1b[32m';

const BROWN = '\x1b[33m';

const RESET = '\x1b[0m';

// Credenciais da Binance

const { apiKey, apiSecret } = workerData;

// Validação das credenciais

if (!apiKey || !apiSecret) {

  console.error('[workerExecBinance] Erro: Credenciais incompletas no workerData:', workerData);

  throw new Error('Credenciais da Binance não fornecidas corretamente');

}

// Função para formatar objetos nos logs

function formatObject(obj) {

  return JSON.stringify(obj, null, 2)

    .replace(/"([^"]+)": "([^"]+)"/g, `"$1": ${GREEN}"$2"${RESET}`)

    .replace(/"([^"]+)": (\d+(.\d+)?)/g, `"$1": ${BROWN}$2${RESET}`)

    .replace(/"([^"]+)": null/g, `"$1": null`);

}

// Função para assinar a mensagem com Ed25519

async function signMessage(message) {

  const secretBuffer = Buffer.from(apiSecret, 'base64');

  const key = await crypto.subtle.importKey(

    'pkcs8',

    secretBuffer,

    { name: 'Ed25519' },

    false,

    ['sign']

  );

  const encoder = new TextEncoder();

  const signature = await crypto.subtle.sign(

    { name: 'Ed25519' },

    key,

    encoder.encode(message)

  );

  return Buffer.from(signature).toString('base64');

}

// Função de log com timestamps

let t0;

function logMessage(channel, message) {

  const abs = Date.now();

  const t = t0 ? abs - t0 : 0;

  console.log(`[workerExecBinance - ${channel}] [abs=${abs} t=${t}] ${message}`);

}

let ws;

let portToCoord;

// Conecta ao WebSocket da Binance

async function connectBinanceWebSocket() {

  ws = new WebSocket('wss://ws-api.binance.com:443/ws-api/v3');

  ws.on('open', async () => {

    logMessage('Conexão', 'Conectado ao WebSocket trade API da Binance.');

    const timestamp = Date.now();

    const params = `apiKey=${apiKey}&timestamp=${timestamp}`; // Corrigido '×tamp' para '&timestamp'

    const signature = await signMessage(params);

    const authRequest = {

      id: 1,

      method: 'session.logon',

      params: {

        apiKey: apiKey,

        timestamp: timestamp,

        signature: signature

      }

    };

    ws.send(JSON.stringify(authRequest));

  });

  ws.on('message', (data) => {

    let msg;

    try {

      msg = JSON.parse(data);

      logMessage('Geral', `Mensagem recebida:\n${formatObject(msg)}`);

    } catch (e) {

      logMessage('Erro', `Erro ao parsear mensagem: ${data}`);

      return;

    }

    if (msg.id === 1 && msg.status === 200) {

      logMessage('Conexão', 'Autenticado com sucesso.');

      portToCoord.postMessage({ type: 'conexoesProntas' });

      // Após autenticação, esperar 5 segundos e enviar a ordem

      setTimeout(async () => {

        const timestamp = Date.now();

        const orderRequest = {

          id: 2,

          method: 'order.place',

          params: {

            price: '84800',

            quantity: '0.0001',

            side: 'SELL',

            symbol: 'BTCUSDT',

            timeInForce: 'GTC',

            timestamp: timestamp,

            type: 'LIMIT'

          }

        };

        logMessage('Postagem', `Enviando ordem:\n${formatObject(orderRequest)}`);

        ws.send(JSON.stringify(orderRequest));

      }, 5000);

    }

  });

  ws.on('error', (err) => {

    logMessage('Erro', `Erro no WebSocket: ${err.message}`);

  });

  ws.on('close', () => {

    logMessage('Conexão', 'WebSocket desconectado. Tentando reconectar em 5s...');

    setTimeout(connectBinanceWebSocket, 5000);

  });

}

// Inicialização

parentPort.on('message', (msg) => {

  if (msg.type === 'init') {

    portToCoord = msg.port;

    logMessage('Geral', 'Inicializando workerExecBinance.');

    t0 = Date.now(); // Define t0 no início para rastreamento de tempo

    connectBinanceWebSocket();

  }

});

**//#####################################################**

Codigo do Modelo:

"use strict";

const WebSocket = require('ws');

const crypto = require('crypto');

console.log('\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*');

// Chaves API de teste

const apiKey = 'nYZcNg7kBZfDAGuKpSOceL1h1YUMzSSwRGZut3KUy32KdpYPB0JKi1TV8liqvkVQ';

const apiSecret = 'MC4CAQAwBQYDK2VwBCIEICtYrvoheU+SozG67W4syoWymQ7Z2bTkgvd+IDypO3AM';

// Função para assinar a mensagem com Ed25519

async function signMessage(message) {

  const secretBuffer = Buffer.from(apiSecret, 'base64');

  const key = await crypto.subtle.importKey(

    'pkcs8',

    secretBuffer,

    { name: 'Ed25519' },

    false,

    ['sign']

  );

  const encoder = new TextEncoder();

  const signature = await crypto.subtle.sign(

    { name: 'Ed25519' },

    key,

    encoder.encode(message)

  );

  return Buffer.from(signature).toString('base64');

}

// Conexão ao WebSocket

const ws = new WebSocket('wss://ws-api.binance.com:443/ws-api/v3');

ws.on('open', async () => {

  console.log('Conectado ao WebSocket trade API!');

  // Autenticação (session.logon)

  const timestamp = Date.now();

  const params = `apiKey=${apiKey}&timestamp=${timestamp}`;

  const signature = await signMessage(params);

  const authRequest = {

    id: 1,

    method: 'session.logon',

    params: {

      apiKey: apiKey,

      timestamp: timestamp,

      signature: signature

    }

  };

  ws.send(JSON.stringify(authRequest));

});

ws.on('message', async (data) => {

  const msg = JSON.parse(data);

  console.log('Mensagem recebida:', msg);

  // Após autenticação bem-sucedida, enviar a ordem

  if (msg.id === 1 && msg.status === 200) {

    console.log('Autenticado com sucesso! Enviando ordem de margem cruzada...');

    const timestamp = Date.now();

    const orderRequest = {

      id: 2,

      method: 'order.place',

      params:{

        price: '83800',

        quantity: '0.0001',

        side: 'BUY',

        symbol: 'BTCUSDT',

        timeInForce: 'GTC',

        timestamp: timestamp,

        type: 'LIMIT',

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

 \* Pelo que entendemos ate agora, nao existe um canal

 \* de ws para postagem de ordens com margem. A unica

 \* forma de fazer isso é atravez de REST. A informacao

 \* veio por uma resposta no forum dos desenvolvedores.

 \*

 \* Para determinar uma operacao em margem a conta de

 \* margin deve estar configurada e ter fundos. A ord

 \* em no REST deve conter estes dois campos:

 \*    isIsolated: 'FALSE', (issolada/cruzada)

 \*    sideEffectType: 'AUTO\_BORROW',

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

      }

    };

    ws.send(JSON.stringify(orderRequest));

  }

});

ws.on('error', (error) => console.error('Erro:', error));

ws.on('close', () => console.log('Conexão fechada'));

// Encerramento manual (Ctrl+C)

process.on('SIGINT', () => {

  console.log('Encerrando...');

  ws.close();

  process.exit(0);

});

**//#####################################################**