DeFi

Decentralized Finance

Thierry Sans

TradFi (Traditional Finance) **vs.** DeFi (Decentralized Finance)

TradFi (Traditional Finance)

- Centralized
- Many intermediaries (brokers, marker makers, clearing houses, exchanges)
- More or less transparency (depends on regulations and oversight)
- KYC (Know Your Customer)

DeFi (Decentralized Finance)

- Decentralized
- No intermediaries (from wallet to smart contract exchanges directly)
- Fully transparent (data and code written in clear on the blockchain)
- Permissionless

Bridging DeFi and TradFi

Two operations bridge the gap between DeFi and TradFi

- **On-Ramp**
  Purchasing cryptocurrency with actual FIAT* currency
- **Off-Ramp**
  Selling cryptocurrency to get actual FIAT* currency

* FIAT is government-issued currency like CAD, USD, EURO

DeFi Summary

- Decentralized Exchanges (a.k.a Automated Market Makers)
- DeFi Staking
- Price Discovery Through Arbitrage
- Borrowing and Lending
- Flash Loans
- Stablecoins
- Yield Farming

Decentralized Exchanges

a.k.a Automated Market Makers

[Recap] ERC-20 Tokens

A fungible token is a smart contract that maintains all user balances

```
mapping(address => uint256) balances
```

ERC-20 is a standard API for fungible tokens

```
function transfer(address to, uint256 value) returns (bool)
function transferFrom(address from, address to, uint256 value) returns (bool)
function approve(address spender, uint256 value) returns (bool)

function totalSupply() view returns (uint256)
function balanceOf(address owner) view returns (uint256)
function allowance(address owner, address spender) view returns (uint256)
```

Concept of Exchange (a.k.a market maker)

An exchange converts one currency to another given an exchange rate

How is the exchange rate defined ?

- It can be set arbitrary (unfair market)
- It can be calculated dynamically based on supply/demand (fair market)

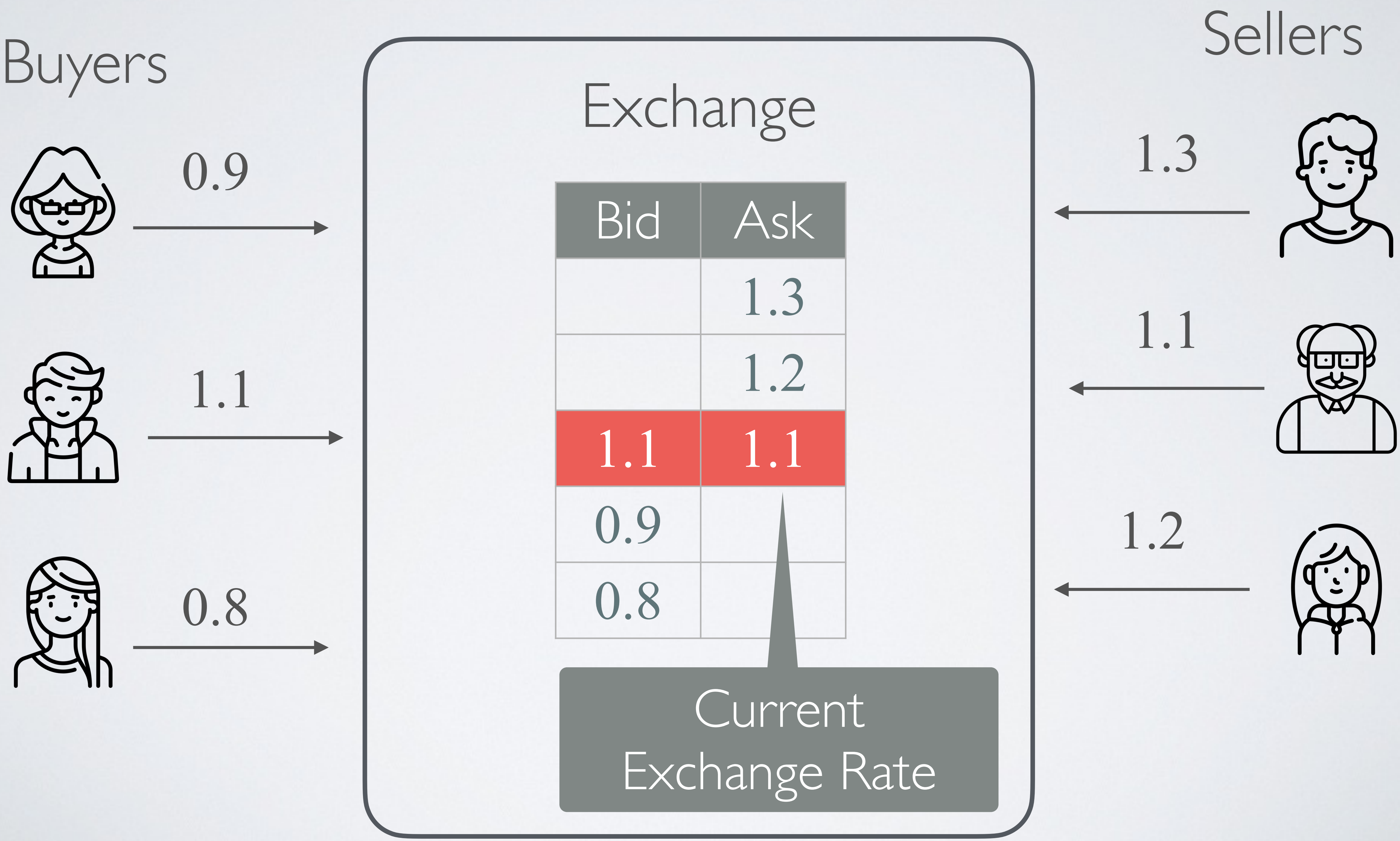
But how to calculate an exchange rate dynamically?

- some people wants to sell token T_A to buy T_B
- other people wants to sell token T_B to buy T_A

➡ an exchange connects buyers and sellers together

✓ The exchange rate is the result of this dynamic process

Order Book-based approach (commonly used in TradFi)



CEX - Centralized Exchange

Can you create a platform that implements an automated order book-based exchange?

➡ Yes, build a platform with :

- a backend to collect user orders
- a wallet to collect money (escrow transaction)

◉ Limitation : centralized approach!
Users must trust the backend to do the right thing and to be always available

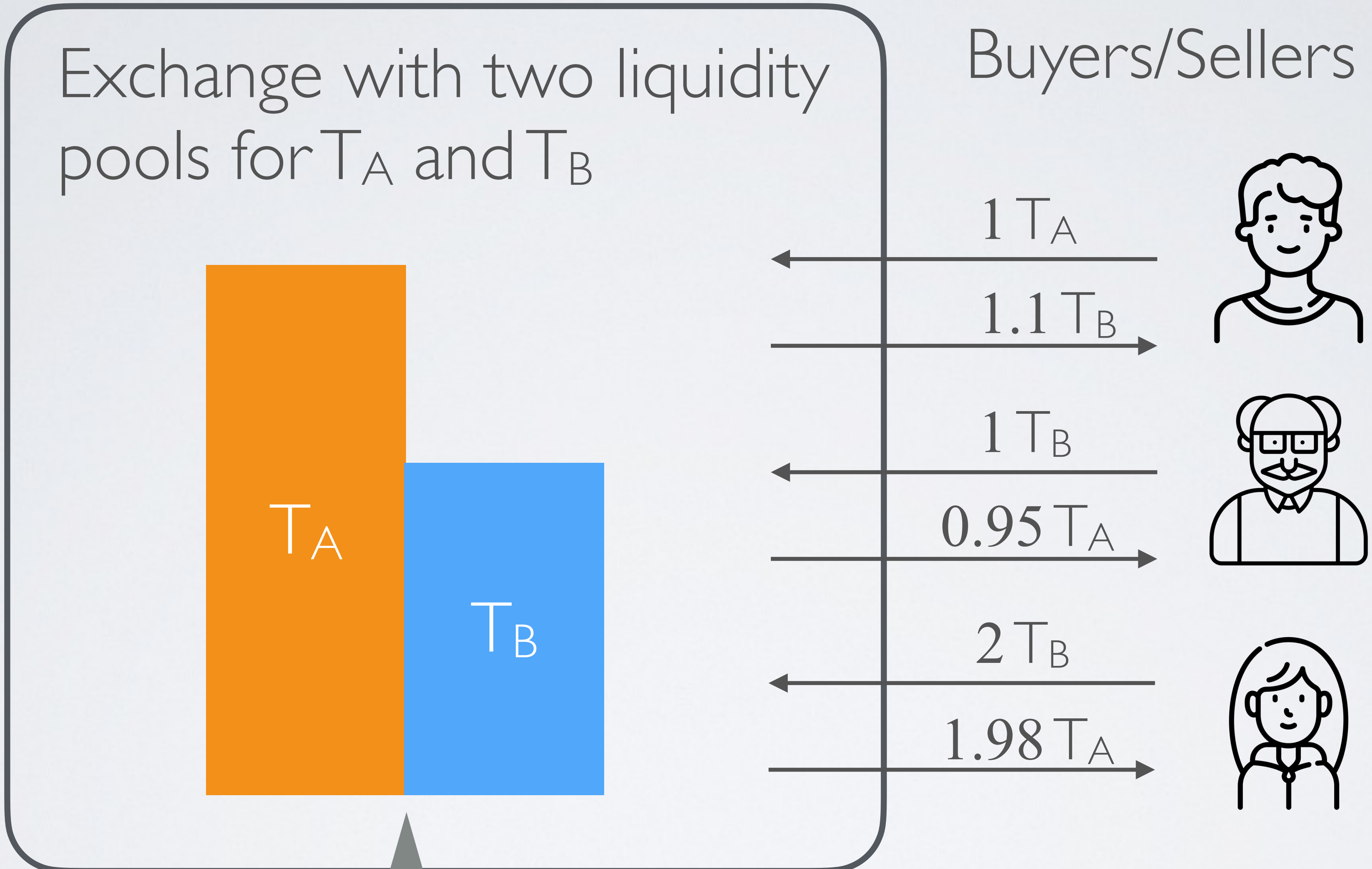
Decentralized version of an order book-based exchange

Can you write a smart contract that implements an automated order book-based exchange?

➡ Technically yes but **no in practice** because of gas

- Matching (i.e ordering) buy/sell orders is expensive
- Placing, withdrawing and fulfilling orders is expensive

A Better Approach : Liquidity Pools



Current Exchange Rate is calculated based on the pool levels

Dynamic pricing using the Constant Product Market Maker

The Constant Product Market Maker is about maintaining a value **k** constant between two liquidity pools

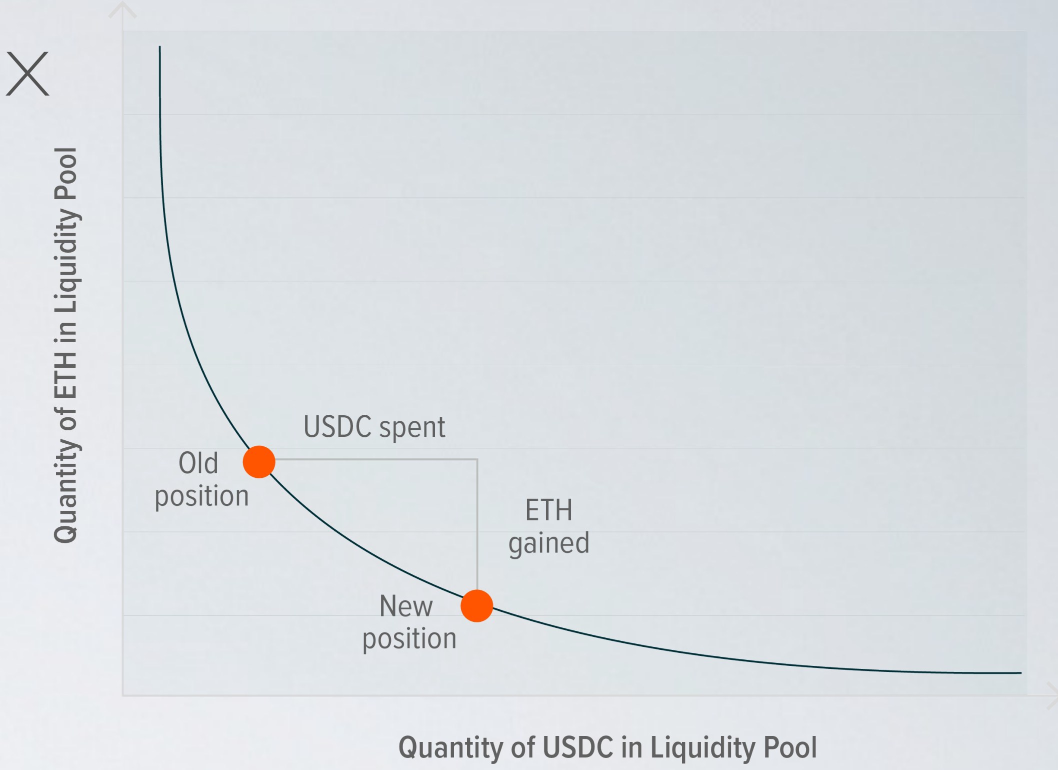
$$\mathbf{k} = \mathbf{vol}(T_A)\ \mathbf{x}\ \mathbf{vol}(T_B)$$

➡ Using this constant, we can calculate swap values
i.e what quantity of T_B must be withdrawn when adding adding a given quantity of T_A to keep **k** constant (and vice versa)

✓ This determines the exchange rate

source: Global X

Calculating the exchange rate



- Swapping **a** amount of T_A for **b** amount of T_B

  Since $vol(T_A) \times vol(T_B) = (vol(T_A) + a) \times (vol(T_B) - b)$

  Then $b = (a \times vol(T_B)) / (a + vol(T_A))$

A liquidity pool cannot be emptied

- Swapping **b** amount of T_B for **a** amount of T_A

  Since $vol(T_A) \times vol(T_B) = (vol(T_A) - a) \times (vol(T_B) + b)$

  Then $a = (b \times vol(T_A)) / (b + vol(T_B))$

Example of exchange rate evolution

Swap Order	Out	Rate (T_B/T_A)	vol(T_A)	vol(T_B)	k
			12	10	120
4 T_A	2.5 T_B	0.625	16	7.5	120
4 T_A	1.5 T_B	0.375	20	6	120
4 T_B	8 T_A	0.5	12	10	120
2 T_B	2 T_A	1	10	12	120
6 T_A	4.5 T_B	0.75	16	7.5	120

Side Effects of the Dynamic Pricing

When a user submits an exchange transaction,
the actual exchange may varies depending on other transactions executed before

➡ Slippage (common to TradFi and DeFi)
Difference between the exchange rate when quoted and the actual one when executed

✓ Most DEXes implement a slippage protection mechanism that allow users to specify a slippage tolerance limit

➡ Sandwich Attack a.k.a front-running attack (specific to DeFi)
An attacker can monitor transactions in the mempool and emit concurrent transactions that will take advantage of dynamic exchange rate

✓ Some DEXes implement swap protection mechanism (e.g *Uniswap* v4)

Slippage Example

Beforehand

- When Alice wants to do a swap, the pool contains 12 T_A and 10 T_B
- So she gets quoted that swapping 4 T_A will get her 2.5 T_B (0.62 exchange rate)
- She submits the swap transaction to the mempool **allowing 20% slippage**

Scenario #1

- Bob submits a swap request for **1 T_A** at the same time (i.e within the same block)
- However, Bob's transaction is executed before Alice's changing the levels of liquidity polls and moving the current exchange rate down
- When Alice's request is executed, she actually gets 2.17 T_B (0.54 exchange rate)

✓ The slippage is 15% (0.62 / 0.54 ~ 1.15) and the swap is executed

Scenario #2

- Bob submits a swap request for **2 T_A** at the same time (i.e within the same block)
- When Alice's request is executed, she actually gets 1.90 T_B (0.48 exchange rate)

◉ The slippage is 30% (0.62 / 0.48 = 1.30) and the swap is not executed

Sandwich Attack Example

- Alice submits a transaction to the mempool to swap 4 T_A with a 20% slippage
- Mallory monitors the mempool and **sees Alice's transaction**
- Mallory immediately submits two transactions :
    1. swap 1.2 T_A **(high tip)**
    2. swap 0.9 T_B (low tip)
- Transactions are executed as follows:
    1. Mallory's transaction (high priority) **swaps 1.2 T_A** into 0.9 T_B
    2. Alice's transaction (medium priority) swaps 4 T_A into 2.12 T_B (which is just within the 20% slippage limit)
    3. Mallory's transaction (low priority) swaps 0.9 T_B **into 1.98 T_A**

➡ Mallory **pockets 0.78 T_A** (risk free) with two simple transactions

Simple DEX Example

```
    function swap(address _fromToken, uint256 _amountIn) external returns (uint256 amountOut) {
        require(_amountIn > 0, "Amount must be greater than zero");
        require(_fromToken == address(token1) || _fromToken == address(token2), "Invalid token");

        bool isToken1 = _fromToken == address(token1);
        IERC20 from = isToken1 ? token1 : token2;
        IERC20 to = isToken1 ? token2 : token1;
        uint256 reserveIn = isToken1 ? reserve1 : reserve2;
        uint256 reserveOut = isToken1 ? reserve2 : reserve1;

        // deduct the fee from in
        uint256 amountMinusFee = (_amountIn * (FEE_DIVISOR - FEE_PERCENT)) / FEE_DIVISOR;

        // calculate the amount of token to swap out
        amountOut = (amountMinusFee * reserveOut) / (reserveIn + amountMinusFee);

        // update the reserves
        if (isToken1) {
            reserve1 += _amountIn;
            reserve2 -= amountOut;
        } else {
            reserve2 += _amountIn;
            reserve1 -= amountOut;
        }
        // transfer the tokens from user to contract
        require(from.transferFrom(msg.sender, address(this), _amountIn), "Swap transfer in failed");
        // transfer the tokens from contract to user
        require(to.transfer(msg.sender, amountOut), "Swap transfer out failed");
    }
```

Deduct the fees

Calculate the swap

Update the pools

Transfer the tokens

DEX Staking

Incentive for Liquidity Pool Providers

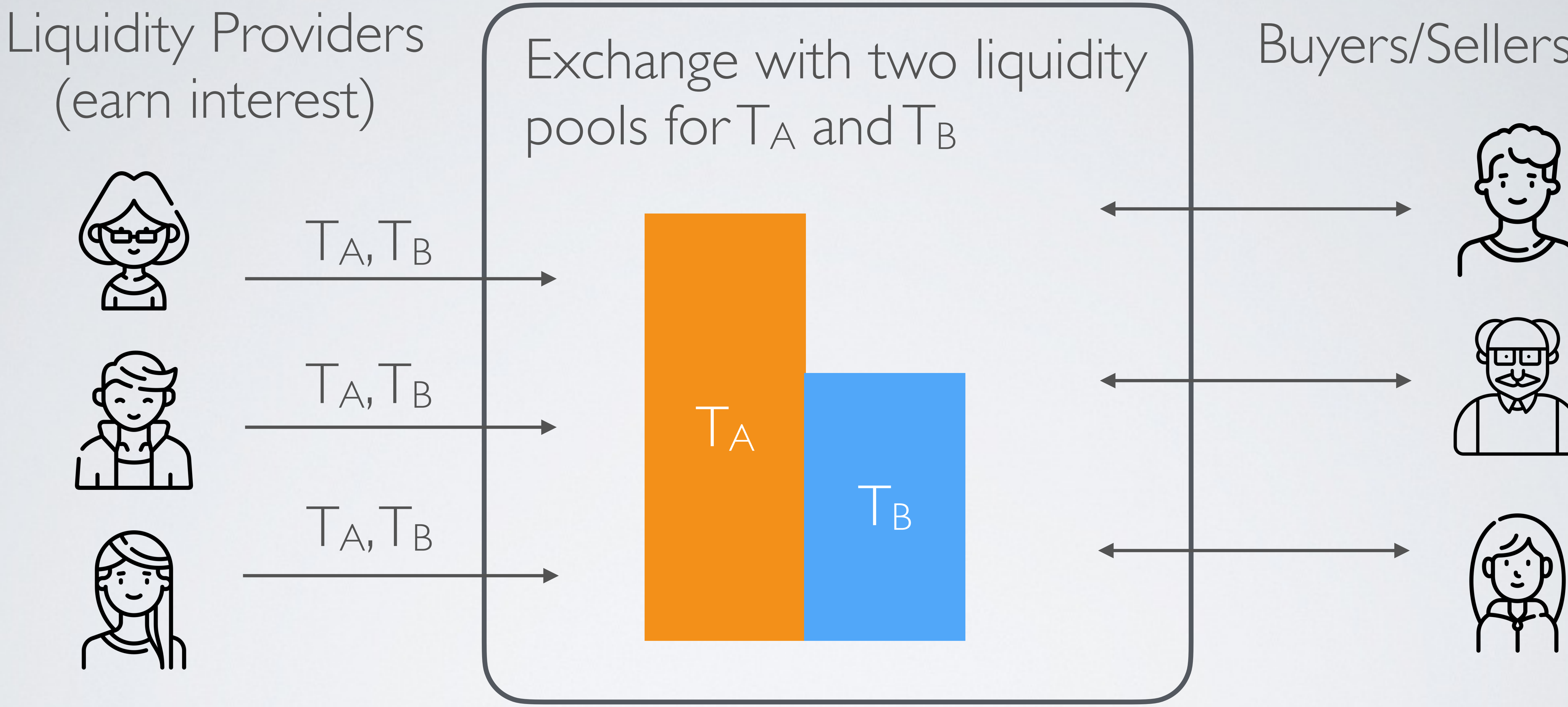
To be efficient, liquidity pools must stash large volumes of cryptocurrencies to absorb a great quantity of possibly unbalanced transactions

So what is the incentive to stake into DEX liquidity pools?

➡ Have a fee for every swap transaction and reward liquidity pool providers (0.30% on *Uniswap* for instance)

◉ This is "DEX Staking"
(not be confused with "Consensus Layer Staking" in PoS)

Liquidity Providers



➡ The liquidity providers provide both T_A and T_B and are rewarded from fees collected on every swap

Liquidity Token

Liquidity providers must **add or withdraw T_A and T_B** from/to the pools **while preserving the ratio** (i.e the exchange rate)

➡ Have a token T_L that represents the contribution to the liquidity pools

✓ **`addingLiquidity`** mints T_L tokens representing the amount of of tokens T_A and T_B deposited by the user to the pools

✓ **`removeLiquidity`** burns T_L tokens allowing the user to withdraws the corresponding amount of tokens T_A and T_B (plus interest generated from fees collected during staking period)

Example

Call	Returns	vol(T_A)	vol(T_B)	real(T_A)	real(T_B)
addLiq(12, 10)	14,400 T_L	12	10	12	10
addLiq(6, 5)	7,200 T_L	18	15	18	15
swap(2.06 T_A)	1.5 T_B	20	13.5	20.06	13.5
swap(4.635 T_B)	5 T_A	15	18	15.06	18.135
rmLiq(7,200 T_L)	5.02 T_A 6.045 T_B	10	12	10.04	12.09

```
    function addLiquidity(uint256 amount1, uint256 amount2) external {
        require(amount1 > 0 && amount2 > 0, "Amounts must be greater than zero");

        uint256 correctAmount2;
        uint256 liquidityMinted;
        // check if the reserves are empty
        if (reserve1 == 0 && reserve2 == 0) {
            // if empty, the amount of token 1 and 2 set pool ratio (a.k.a the exchange rate)
            correctAmount2 = amount2;
            // and the amount of lpToken to mint is (amount1* amount2)^2
            liquidityMinted = amount1 * amount2 * amount1 * amount2;
        } else {
            // calculate the right amount of token2 to add to preserve the liquidity pool ratio
            correctAmount2 = (amount1 * reserve2) / reserve1;
            require(amount2 >= correctAmount2, "Insufficient token2 amount provided");
            amount2 = correctAmount2;
            // calculate the amount of lpToken to mint
            liquidityMinted = amount1 * lpToken.totalSupply() / reserve1;
        }

        uint256 amount1ToPay = amount1;
        uint256 amount2ToPay = correctAmount2;

        uint256 token1Reward;
        uint256 token2Reward;
        (token1Reward, token2Reward) = calculateReward(msg.sender);
        amount1ToPay -= token1Reward;
        amount2ToPay -= token1Reward;

        // mint the lpToken
        lpToken.mint(msg.sender, liquidityMinted);
        isStaking[msg.sender] = true;

        // update rewardPerTokenPaid
        rewardPerToken1Paid[msg.sender] = rewardPerToken1;
        rewardPerToken2Paid[msg.sender] = rewardPerToken2;

        // update the reserves
        reserve1 += amount1;
        reserve2 += correctAmount2;

        // transfer the funds from user to contract
        require(token1.transferFrom(msg.sender, address(this), amount1ToPay), "Token1 transfer failed");
        require(token2.transferFrom(msg.sender, address(this), amount2ToPay), "Token2 transfer failed");
    }
```

```
    function removeLiquidity(uint256 lpAmount) external {
        require(lpAmount > 0, "Invalid LP token amount");
        require(lpToken.balanceOf(msg.sender) >= lpAmount, "Insufficient LP balance");

        // calculate the amounts of token1 and token 2
        uint256 totalSupply = lpToken.totalSupply();
        uint256 amount1 = (lpAmount * reserve1) / totalSupply;
        uint256 amount2 = (lpAmount * reserve2) / totalSupply;

        uint256 token1Reward;
        uint256 token2Reward;
        (token1Reward, token2Reward) = calculateReward(msg.sender);

        // update rewardPerTokenPaid
        rewardPerToken1Paid[msg.sender] = rewardPerToken1;
        rewardPerToken2Paid[msg.sender] = rewardPerToken2;

        // update the reserves
        reserve1 -= amount1;
        reserve2 -= amount2;

        // burn the lpTokens
        lpToken.burn(msg.sender, lpAmount);
        isStaking[msg.sender] = (lpToken.balanceOf(msg.sender) > 0);

        // transfer the funds from contract to user
        require(token1.transfer(msg.sender, amount1 + token1Reward), "Token1 transfer failed");
        require(token2.transfer(msg.sender, amount2 + token2Reward), "Token2 transfer failed");
    }
```

The risk Behind Staking a.k.a Impermanent Loss

When liquidity provider deposits 10 T_A and 5 TB, the exchange rate is 2

➡ So the amount deposited is worth **20 T_A when staking**

When the liquidity providers withdraws the whole stake for 14 T_A and 3.5 T_B and the exchange rate is 4

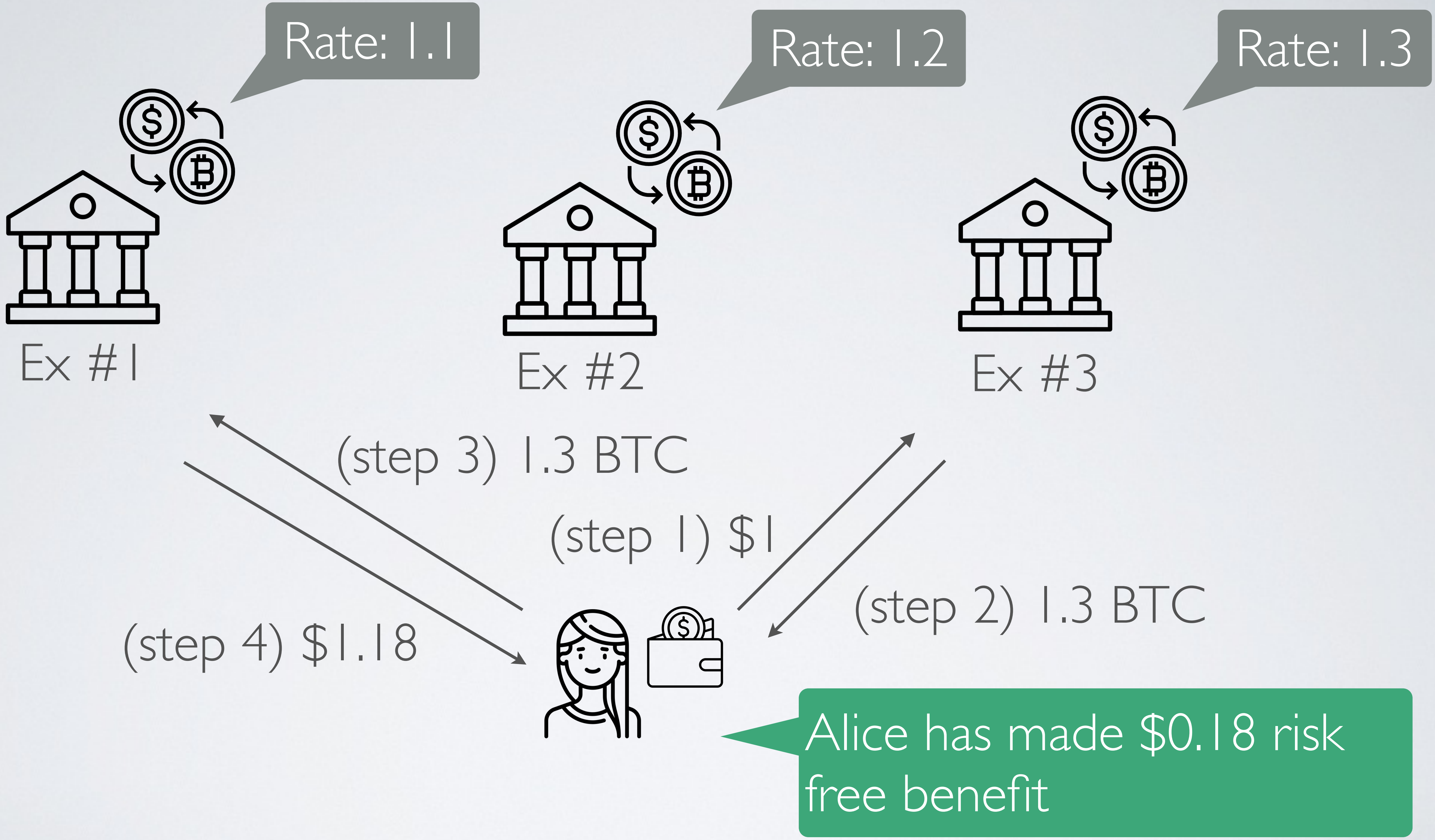
➡ This amount is worth **28 T_A after withdrawing**

However, the **initial stake is now worth 30 T_A without staking**

- This is an **impermanent loss**
  It is an unrealized loss until the liquidity provider withdraws its stake

Automatic Price Discovery

In a decentralized world



➡ Alice can repeat the operation over and over while there are exchanges with different rates

The concept of Arbitrage

Arbitrage (common to TradFi and DeFi)

Exploiting price differences between markets

➡ Traders do take advantage of rate differences between exchanges to make risk-free profits

Automatic Price Discovery resulting from Arbitrage

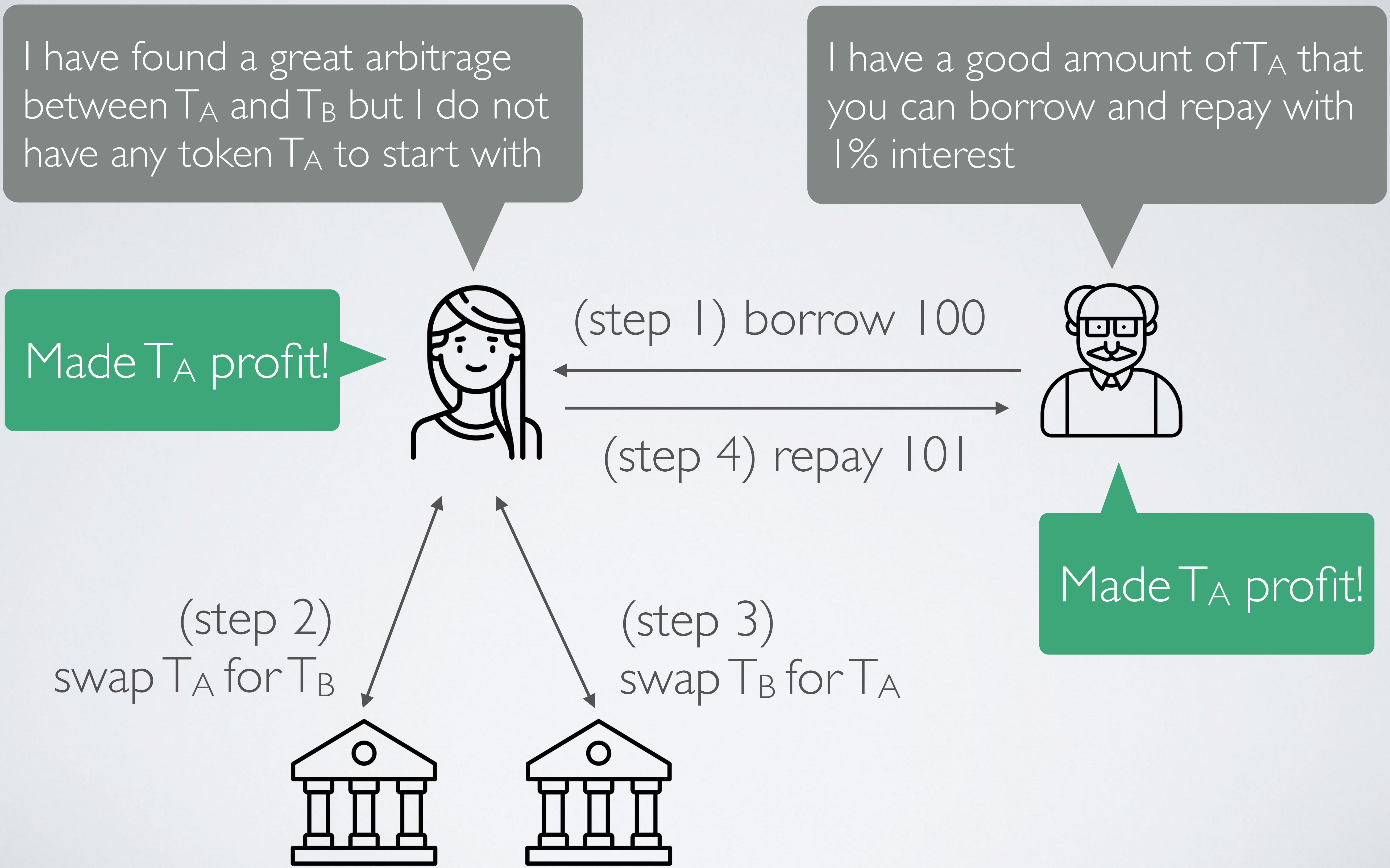
As traders take advantages of arbitrage, the market as a whole move to a state where no one can make these profits (also called *Nash Equilibrium*)

➡ All exchanges converge to the same rate
a.k.a the market price

✓ This process is called **automatic price discovery**

Flash Loans

Lending and Borrowing



Concept of Flash Loan

◉ Problem : how to make sure that the borrower will repay the loan?

✓ Solution : borrow and repay the token **within the same transaction**

➡ **Flash Loan**

Smart contract aggregator

Borrow token

```
    function borrow(uint256 amount, Call[] memory calls) external {
        require(token.transfer(msg.sender, amount), "Token transfer failed");
        for (uint256 i = 0; i < calls.length; i++) {
            (bool success, ) = calls[i].target.call(calls[i].callData);
            require(success, "Call failed");
        }
        uint256 amountWithInterest =  amount * (RATE_DIVISOR + INTEREST_RATE) / RATE_DIVISOR;
        require(token.transferFrom(msg.sender, address(this), amountWithInterest), "Token transfer failed");
    }
```

Execute other transactions (arbitrage for instance)

Repay token

If the token is not repaid (with interest) , the last transaction fails

✓ Then all other transactions including borrowing the token fails as well

➡ This is **risk free** for the lender

Arbitrage Example

Borrower has no token

```
it("Should do a flash loan arbitrage", async function () {
    expect(await token1.balanceOf(borrower.address)).to.be.eq(0);

    await token1.connect(borrower).approve(await dex1.getAddress(), 100);
    const call1 = {
      target: await dex1.getAddress(),
      callData: dex1.interface.encodeFunctionData("swap", [borrower.address, (await token1.getAddress()), 100])
    };

    await token2.connect(borrower).approve(await dex2.getAddress(), 125);
    const call2 = {
      target: await dex2.getAddress(),
      callData: dex2.interface.encodeFunctionData("swap", [borrower.address, (await token2.getAddress()), 125])
    };

    await token1.connect(borrower).approve(await flash.getAddress(), 101);
    await flash.connect(borrower).borrow(100, [call1, call2]);

    expect(await token1.balanceOf(borrower.address)).to.be.gt(0);
});
```

Swap 1

Swap 2

Borrow 100 to do swaps 1 & 2

Borrower has made token profit

Borrowing and Lending

(non-flash) Borrowing and Lending

Is it possible to borrow money now (block **n**) and repay it with interest later (block **n+i**) ?

- Problem : we need to make sure that the borrower repays the loan (somehow) before the *maturity date*

Concept of Lending Market

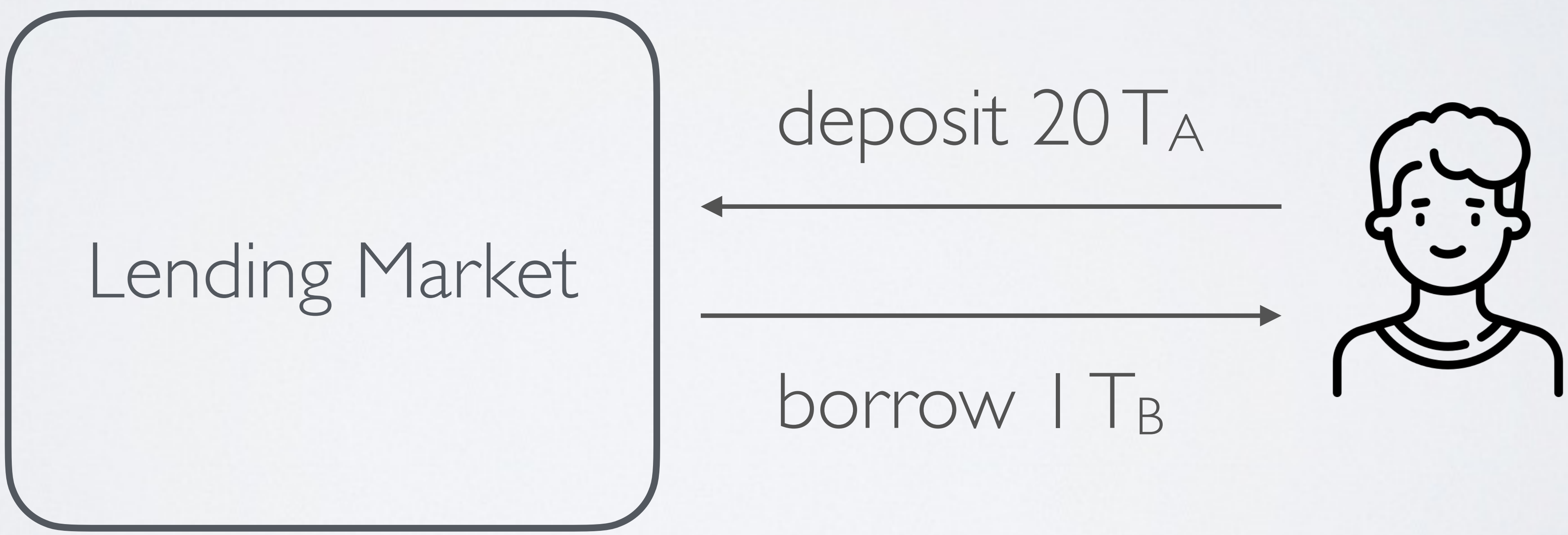
Lenders make money available to borrow and earn interest

Borrowers can take out a loan and repay it with interest before maturity date

The concept of collateral

What if a borrower defaults on a loan?

➡ When taking out a loan, the lender must deposit a **collateral** i.e assets that serve as a security deposit



Undercollaterized vs overcollaterized

Undercollaterized
Borrower must provide a collateral that is less than the value of the loan

$$\text{val(collateral)} < \text{val(loan)}$$

Overcollaterized
Borrower must provide a collateral that is more than the value of the loan

$$\text{val(collateral)} > \text{val(loan)}$$

- Problem : over time the value of the loan might increase compared to the value of the collateral (because of exchange rate)
  - ➡ An undercollaterized loan might become further undercollaterized
  - ➡ An overcollaterized loan might become undercollaterized

Collateral factor

Throughout the whole loan period, the value of collateral must not fall below a threshold called **collateral factor**

➡ i.e $\mathrm{val(collateral)} \times k > \mathrm{val(loan)}$ must remain true

This factor is usually

- >1 for undercollaterized loan
- <1 for overcollaterized loan

The role of collateral

After taking a loan, two things can happen:

1. The borrower repays the loan with interest
   ➡ Collateral is returned

2. The borrower defaults (i.e does not repay before maturity date)
   ➡ The loan is **liquidated** (i.e repaid with collateral) with a penalty

3. Before maturity date, the value of collateral falls under
   ➡ The loan is **liquidated** with a penalty

CeFI vs DeFI

CeFi - both undercollaterized and overcollaterized lending schemes are common (enabled by laws and regulations)

- e.g mortgage (overcollaterized)
- e.g credit card (undercollaterized)

DeFi - only overcollaterized lending schemes are common

- e.g *Aave*, *Compound*, *Curve Finance*

Example of Overcollaterized Loan

A borrower deposits **$15, 000 USDC** as collateral to borrow **5 ETH** (priced at $2,000 USD each) in lending market with:

- Collateralization required: 1.5 (150%)
- Collateralization factor: 1.2 (120%)
- Penalty: 0.1 (10%)
- Interest rate: 0.05 (5%)

What can happen:

- Either the borrower repays 5,25 ETH and $15,000 USDC are returned
- Or the borrower defaults when 1 ETH is $2,200 USDC, $12,100 USDC (worth 5.5 ETH) are liquidated and $2,900 USDC are returned to the borrower
- Or ETH rises above $2,400 (10,000*1.2/5), $13,200 USDC are liquidated and $1,800 USDC are returned to the borrower

Why overcollateralized borrowing makes sense

Why taking an overcollaterized loan when you can just exchange the collateral

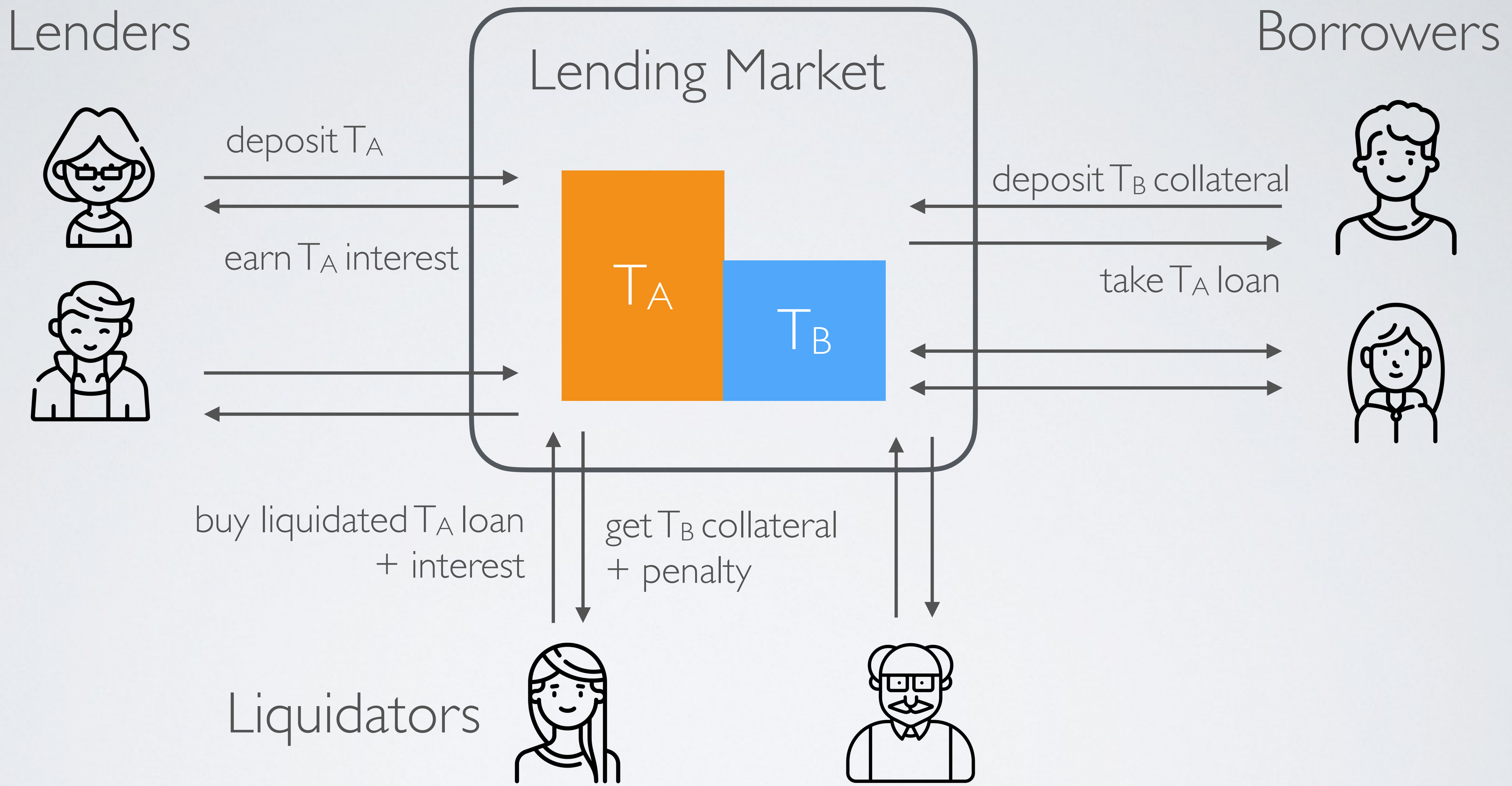
➡ In a nutshell : **shorting**
i.e betting that the value of an asset will go down

Example

1. Borrow 5 ETH (1 ETH - $2,000) with $15,000 collateral
2. and sell it right away to cash out $10,000
3. Wait for ETH to decrease to $1,500 and buy back 5.5 ETH for $7,875
4. Repays 5.25 ETH and get the full collateral back

✓ The borrower pockets $2,125

Implementing a DeFi Lending Market



➡ The liquidators are important to make sure the T_A pool does dry up when borrowers are defaulting

Linear Variable Interest Rate

When the **supply of T_A is low**, raise interest rate to

- decrease the demand of loans
- and increase the incentive for lenders to stake T_A

➡ Have a interest rate that depends on the utilization of the T_A pool

Utilization ratio $U = \text{totalBorrow} / \text{totalDeposit}$

Borrow rate $R = \text{baseRate} + U$

✓ The interest rate varies on every deposit, loan and liquidation transactions

Slopped Variable Interest Rate

Used in common DeFI Lending Markets such as AAVE and Compound

Same idea but the goal is to reach an optimal pool utilization (usually 80% by empirical model)

➡ The model takes into account how fast the interest rate should varies (a.k.a **slope**) to reach the optimal utilization rate

- **Slope1** defines how the borrow rate increases as utilization rises up to the optimal utilization rate (to incentivize borrowing)
- **Slope2** Defines how sharply the borrow rate increases after the optimal utilization point (to penalize borrowing)

$$\text{Borrow Rate} = \begin{cases} \text{Base Rate} + U \times \text{Slope}_1, & \text{if } U \leq U_{opt} \\ \text{Base Rate} + \text{Slope}_1 + \frac{U - U_{opt}}{1 - U_{opt}} \times \text{Slope}_2, & \text{if } U > U_{opt} \end{cases}$$

Stablecoins

Concept of Stablecoin

A cryptocurrency designed to trade at a fixed price

a.k.a **pegged price**

1 coin = 1 USD

Goals

- Integrate real-world currencies into on-chain applications
- Enable people without easy access to USD, to hold and trade a USD-equivalent asset

Peg Stabilization Mechanism

Maintaining the peg means maintaining the exchange rate between the stablecoin (SC) and its value (USD)

➡ If SC loses its peg, the goal is to create arbitrage opportunities to restore the peg

- If it trades **above its peg** (e.g., 1.05 USD per SC), users can mint stablecoins cheaply and sell them at a profit
- If it trades **below its peg** (e.g., 0.95 USD per SC), users can buy SC on the market and repay their debt at a discount

Centralized vs decentralized stablecoins

Centralized stablecoins can be:

- overcollaterized such as *USDC*, *USDT*
- undercollaterized such as some of the Central-Bank Digital Currency (CBDC)

➡ Relies on trusted-third party to maintain the peg

Decentralized stablecoins are necessarily overcollaterized such as *DAI* (MakerDAO)

➡ Relies on smart-contract algorithm to maintain the peg

Decentralized Stablecoin

Decentralized Stablecoin are implemented as a Collateralized Debt Position (similar to borrowing schemes)

- Users can borrow (actually mint) SC by depositing collateral
- Users can get collateral back by repaying (actually burn) SC

What is the interest for users?

- take part of trading activities
- maintain liquidity while the value of the collateral could increase
- earn interest while holding (e.g stability fee in DAI)

Minting Stablecoin

```
    function depositCollateral(uint256 amount) external {
        require(amount > 0, "Amount must be greater than zero");
        require(collateralToken.transferFrom(msg.sender, address(this), amount), "Token transfer fail");
        deposits[msg.sender] += amount;
    }

    function mint(uint256 stablecoinAmount) external {
        uint256 collateralValue = deposits[msg.sender] * oracle.getExchangeRate() / RATE_DIVISOR;
        uint256 requiredCollateral = (stablecoinAmount * COLLATERALIZATION_RATIO) / RATE_DIVISOR;
        require(collateralValue >= requiredCollateral, "Insufficient collateral");
        stableToken.mint(msg.sender, stablecoinAmount);
    }
```

Overcollaterization

If SC trades **above its peg**,
users can mint SC cheaply and sell them at a profit

Burning Stablecoin

```
    function burn(uint256 stablecoinAmount) external {
        require(stableToken.balanceOf(msg.sender) >= stablecoinAmount, "Insufficient balance");
        stableToken.burn(msg.sender, stablecoinAmount);
    }

    function withdrawCollateral(uint256 amount) external {
        uint256 exchangeRate = oracle.getExchangeRate();
        uint256 collateralValue = deposits[msg.sender] * exchangeRate / RATE_DIVISOR;
        uint256 stablecoinDebt = stableToken.balanceOf(msg.sender);
        uint256 requiredCollateral = (stablecoinDebt * COLLATERALIZATION_RATIO) / RATE_DIVISOR;
        require(collateralValue - (amount * exchangeRate / RATE_DIVISOR) >= requiredCollateral, "Undercollateralized");
        deposits[msg.sender] -= amount;
        require(collateralToken.transfer(msg.sender, amount), "Token transfer fail");
    }
```

If SC trades **below its peg**,
users can buy SC on the market and repay their debt at a discount

Liquidation
as an additional incentive to burn

```
    function liquidate(address user) external {
        uint256 exchangeRate = oracle.getExchangeRate();
        uint256 collateralValue = deposits[user] * exchangeRate / RATE_DIVISOR;
        uint256 stablecoinDebt = stableToken.balanceOf(user);
        uint256 requiredCollateral = (stablecoinDebt * COLLATERALIZATION_RATIO) / RATE_DIVISOR;
        require(collateralValue < requiredCollateral, "Position is not undercollateralized");
        stableToken.burn(user, stablecoinDebt);
        require(collateralToken.transfer(msg.sender, deposits[user]), "Token transfer fail");
        deposits[user] = 0;
    }
```

If SC trades **below its peg**,
liquidation creates a virtuous circle in which users can buy SC to get more collateral to buy SC to ... until the peg is restored

Yield Optimization

a.k.a yield farming
a.k.a yield optimization

Concept of Yield Optimizer

A yield optimizer takes advantages of DeFi protocols to maximize revenue by doing

- **Multi-protocol and multi-chain optimization**
  By taking advantages of multiple protocols through multiple chains
- **Auto-compounding**
  By automating harvesting rewards and reinvesting them

➡ Concept of **yield optimization strategy** (implemented as a smart contract)

Benefits for Users

- **Saves gas fees**
  Users don't need to manually compound their yields
- **Fund pooling**
  To mutualize assets to increase the yield
- **Maximizes returns**
  Strategies optimize compounding frequency
- **Security & decentralization**
  Smart contracts handle the funds, reducing custodial risks

Anatomy of a Yield Optimizer Contract

Manage assets
Usually implemented as a **vault** (ERC-4626)

```
contract SimpleYieldOptimizer {

    constructor() {
        ...
    }

    function deposit(uint256 amount) external {
        ...
    }

    function withdraw(uint256 amount) external {
        ...
    }

    function harvest() external {
        ...
    }
}
```

Implements the **strategy**
Called by an automator off-chain at optimal intervals

Going Further

CEX vs DEX
(by *CoinMarketCap* in March 2025)

#	Exchange	Trading volume(24h)
1	Binance	$21,898,243,379
2	Bybit	$3,001,354,147
3	Coinbase Exchange	$4,177,804,769
4	OKX	$3,534,845,459
5	Upbit	$2,882,840,442
6	Bitget	$3,240,544,353
7	MEXC	$4,147,782,955
8	KuCoin	$1,040,761,782
9	Gate.io	$3,076,817,249
10	Kraken	$1,488,212,060

#	Name	Trading volume(24h)
1	Uniswap v3 (Ethereum)	$979,390,648
2	dYdX v4	$256,993,388
3	Curve (Ethereum)	$206,536,401
4	Uniswap v2	$132,328,884
5	THORChain	$106,283,253
6	Balancer v2 (Ethereum)	$35,935,737
7	PancakeSwap v3 (Ethereum)	$21,931,694
8	DeGate	$6,342,019
9	SushiSwap (Ethereum)	$2,615,430
10	Solidly (Ethereum)	$2,382,609

- Exchanges are not very decentralized

Top Lending and Borrowing Markets
(by *CoinMarketCap* in March 2025)

Name	Market Cap
Aave AAVE	$3,268,739,511
Maker MKR	$1,118,621,954
Kava KAVA	$488,956,442
Compound COMP	$449,079,364
Morpho MORPHO	$395,917,433

Top Stablecoins
(by *CoinMarketCap* in March 2025)

#	Name	Market Cap
4	Tether USDT	$142,738,775,867
7	USDC USDC	$57,240,707,146
25	Ethena USDe USDe	$5,427,410,168
27	Dai DAI	$5,365,865,616
46	First Digital USD FDUSD	$2,102,765,884

Decentralized

Yield Optimizer

(by *CoinMarketCap* in March 2025)

Name	Market Cap
Convex Finance CVX	$219,105,802
yearn.finance YFI	$186,682,190
Bella Protocol BEL	$67,337,176
Badger DAO BADGER	$60,914,547
THENA THE	$40,214,911
Stella ALPHA	$38,690,813
Inverse Finance INV	$20,240,879
Harvest Finance FARM	$19,253,722
Beefy BIFI	$16,552,946

Other DeFi protocols

Decentralized Derivatives

Decentralized Indices

Decentralized Insurance (e.g. *Nexus Mutual*)