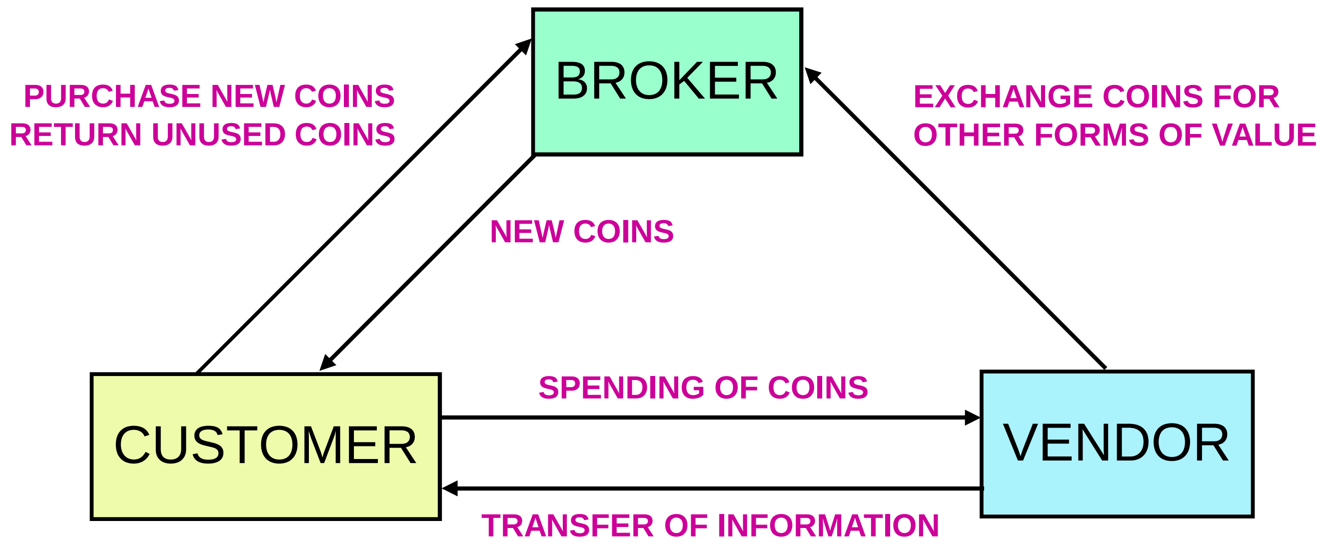
Electronic Payment Systems 20-763

Lecture 9: Micropayments II

Institute for eCommerce

MicroMint

- Brokers produce “coins” having short lifetimes, sell coins to users
- Users pay vendors with coins
- Vendors exchange the coins with brokers for “real” money



SOURCE: SHERIF

Minting Coins in MicroMint

- Idea: make coins easy to verify, but difficult to create (so there is no advantage in counterfeiting)
- In MicroMint, coins are represented by hash-function collisions, values x, y for which H(x) = H(y)
- If H(•) results in an n-bit hash, we have to try about $2^{n/2}$ values of x to find a first collision
- Trying $c \cdot 2^{n/2}$ values of x yields about c^2 collisions
- Collisions become cheaper to generate after the first one is found

Coins as k-way Collisions

- A k-way collision is a set $\{ x_1, x_2, . . ., x_k \}$ with
  $$H(x_1) = H(x_2) = . . . = H(x_k)$$
- It takes about $2^{n(k-1)/k}$ values of x to find a k-way collision
- Trying $c \bullet 2^{n(k-1)/k}$ values of x yields about c^k collisions
- If $k > 2$, finding a first collision is slow, but subsequent collisions come fast
- If a k-way collision $\{ x_1, x_2, . . ., x_k \}$ represents a coin, easily verified by computing $H(x_1), H(x_2), . . ., H(x_k)$
- A broker can easily generate 10 billion coins per month using one machine

Selling MicroMint Coins

- Broker generates 10 billion coins and stores (x, H(x)) for each coin, having a validity period of one month
- The function H changes at the start of each month
- Broker sells coins { x_1, x_2, . . ., x_k } to users for "real" money, records who bought each coin
- At end of month, users return unused coins for new ones

Spending MicroMint Coins

- User sends vendor a coin $\{ x_1, x_2, \ldots, x_k \}$
- Vendor verifies validity by checking that $H(x_1) = H(x_2) = \ldots = H(x_k)$. (k hash computations)
- Valid but double-spent coins (previously used with a different vendor) cannot be detected at this point
- At end of day, vendor sends coins to broker
- Broker verifies coins, checks validity, checks for double spending, pays vendor
- (Need to deal with double spending at this point)

Detecting MicroMint Forgery

- A forged coin is a k-way collision $\{ x_1, x_2, . . ., x_k \}$ under H(•) that was not minted by broker
- Vendor cannot determine this in real-time
- Small-scale forgery is impractical
- Forged coins become invalid after one month
- New forgery can't begin before new hash is announced
- Broker can issue recall before the month ends
- Broker can stay many months ahead of forgers

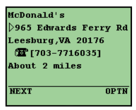
Statistical Schemes

- During World War II, Cola-Cola raised the price of a bottle from 5 cents ($0.05) to 6 cents ($0.06)
- It was expensive to change the coin mechanism
- Coca-Cola randomly removed 1/5 of the bottles from its machines but kept the 5-cent mechanism
- 4/5 of the time a customer would receive a bottle for 5 cents
- 1/5 of the time a customer would pay 5 cents and get NOTHING
- The AVERAGE price of a bottle was 6 cents
- Rarely, a user might pay a lot for a bottle (1 in 625 bottles cost 20 cents)

Statistical Payment

User needs to pay Mapquest $0.01

MAPQUEST



1/1000

999/1000

$10

VOID

Enable 1000 Transactions at Cost of 1

FAIRNESS:

- User, merchant and bank cannot cheat
- Not always fair to user (might be overcharged)
- Fair to merchant and bank on average

MR1 (Micali, Rivest)

- Three parties: user U, merchant M, bank B
- For simplicity, assume every transaction is worth \$0.01 but we only want to process transactions with probability 1/1000
- U and M have public-private key pairs
- Let F be a publicly available function (everyone can obtain the code) that returns a number between 0 and 1 uniformly. (The values of F are uniformly distributed between 0 and 1.)
- A transaction string

  T = User ID || Merchant ID ||Bank ID ||timestamp

SOURCE: RON RIVEST

MR1, continued

- When User U wants to pay Merchant M, he sends M his digital signature C for transaction string T

  $C = Sig_U(T)$ (hash of T encrypted with U's private key)
- Merchant M now signs C

  $D = Sig_M(C)$ (hash of C encrypted with M's private key)
- Merchant M computes F(D)
  - If F(D) < .001, then C is worth $10; otherwise C is worth $0
  - This occurs 1/1000 of the time
- If C has value, M sends to bank C & $D = Sig_M(C)$
- Bank verifies signatures and F(D), charges U $10 and credits M with $10
- No risk to bank; U may pay a lot more than the transaction value

SOURCE: RON RIVEST

Properties of MR1

- Payment is off-line
  - U and M do not have to be in contact during transaction
  - U can send C by email
  - M does not have to contact Bank during transaction
- Bank only sees 0.1% of transactions
- No risk to bank
- Because of signatures, neither U nor M can cheat (if protocol is implemented properly)
- U may pay a lot more than the transaction value
- Want a protocol in which U never pays more than the transaction value

SOURCE: RON RIVEST

MR2 (Micali, Rivest)

- Goal: make sure U never pays more than transaction value he uses
- Shift risk from User to Bank. This is OK because Bank processes large number of transactions
- U includes a serial number S as part of the transaction string
- Let MaxS be the highest serial number the Bank has processed for user U so far (starts at 0)
- When Bank processes a payable transaction:
  - credits M with $10
  - debits U by S – MaxS
  - MaxS $\leftarrow$ S

Why Does MR2 Work?

- User NEVER pays more than the number of transactions he creates
- After *n* transactions, serial number S = *n*. Suppose he has to pay *m* times
- Total payment =

$$\sum_{i=1}^{m} S_i - MaxS_i = \sum_{i=1}^{m-1} MaxS_{i+1} - MaxS_i = S_m = n$$

- User can cheat (by not incrementing S), but Bank will catch him
- Bank on average receives as much as it pay out

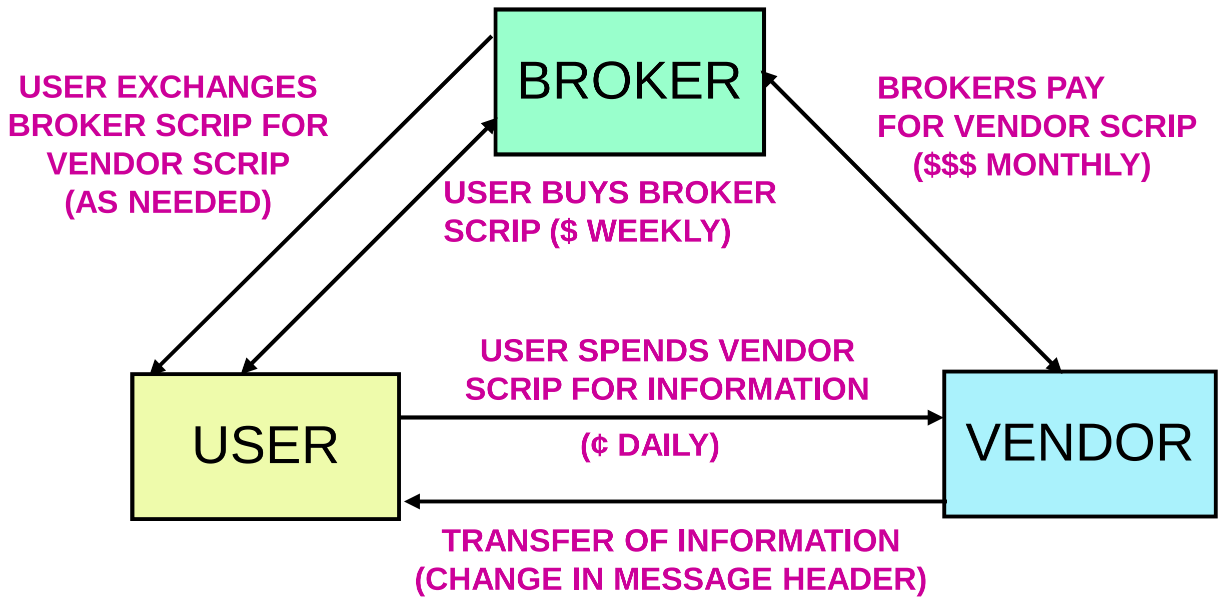
Properties of MR1 and MR2

- Highly scalable: billions of transactions handled with only millions of payments
- Inexpensive
- Payments are offline
- Global aggregation (can handle payments to many merchants from many customers)

SOURCE: RON RIVEST

Millicent

- Vendors produce vendor-specific “scrip”, sell to brokers for “real” money at discount
- Brokers sell scrip from many vendors to many users
- Scrip is prepaid: promise of future service from vendor
- Users “spend” scrip with vendors, receive change



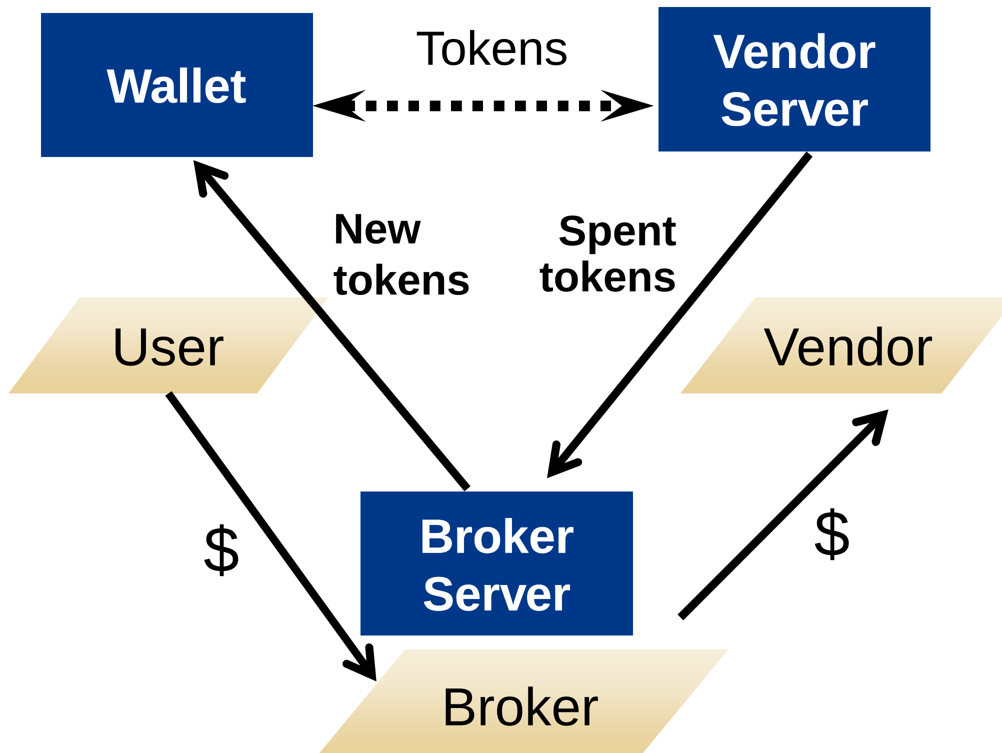
SOURCE: COMPAQ

Millicent

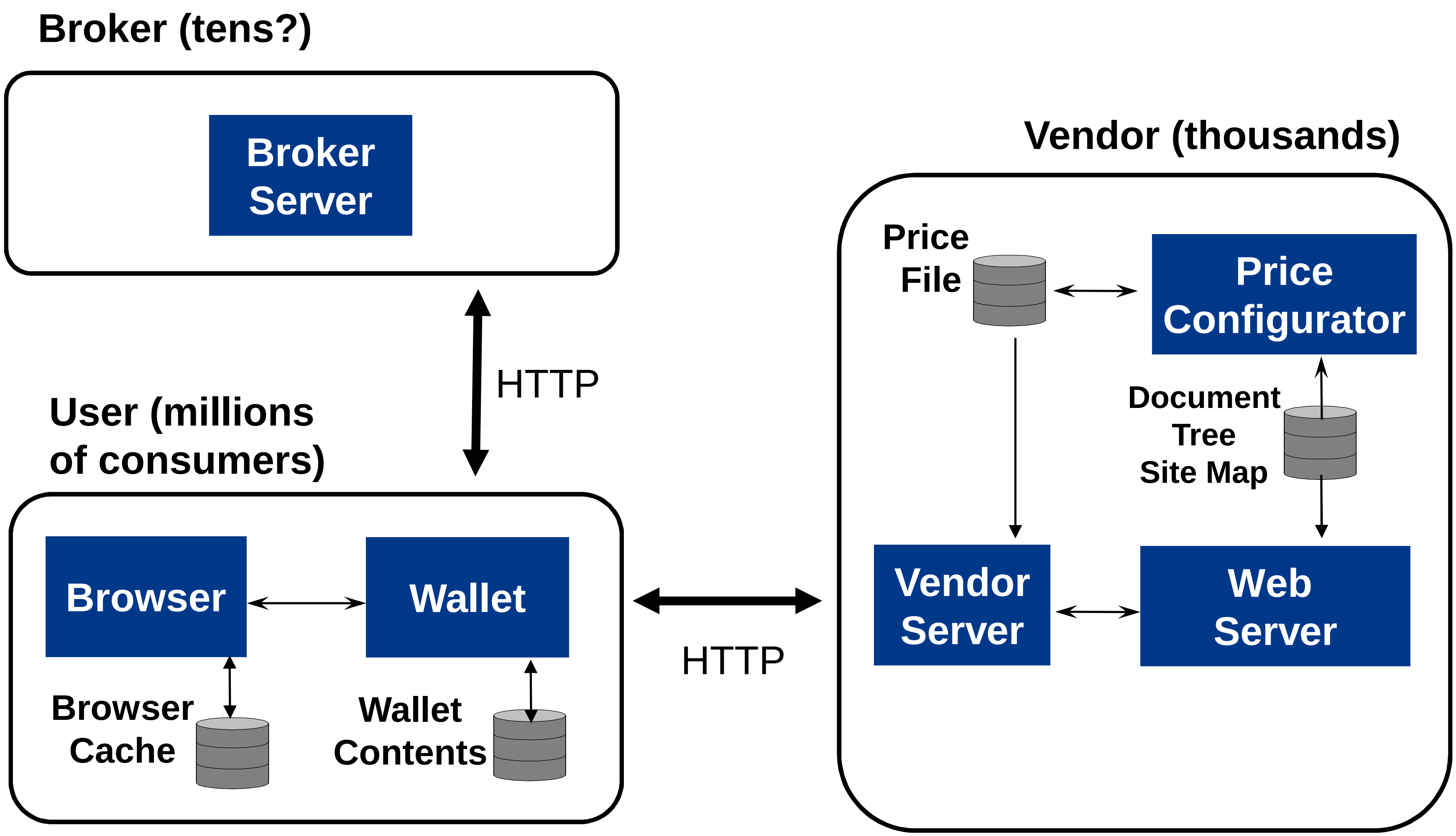
- Broker
  - issues broker scrip to user
  - exchanges broker scrip for vendor scrip
  - interfaces to banking system
  - collects funds from users
  - pays vendors (less commission)
- User
  - buys broker scrip from brokers
  - spends by obtaining vendor-specific scrip from broker
- Vendor
  - sells scrip to brokers
  - accepts vendor scrip from users
  - gives change to users in vendor scrip

MilliCent Components

- **Wallet**
  - integrated with browser as a “proxy”
  - User Interface (content, usage)
- **Vendor software**
  - easy to integrate as a web relay
  - utility for price management
- **Broker software**
  - handles real money

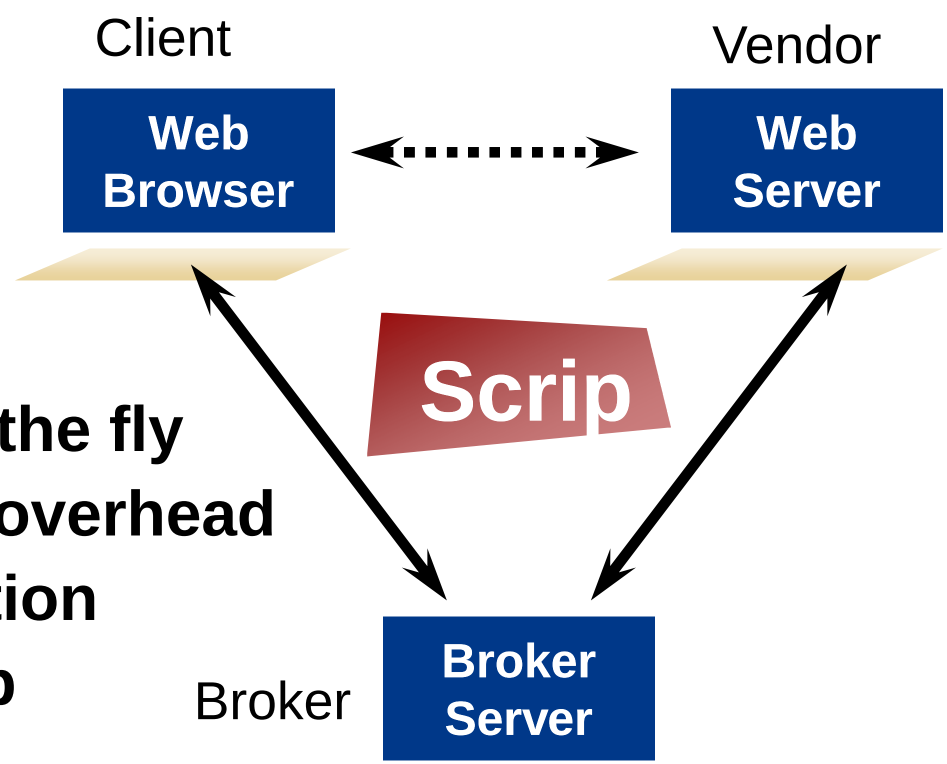


MilliCent System Architecture



Millicent Scrip Verification

- Token attached to HTTP requests
- Scrip can not be:
  - Spent twice
  - Forged
  - Stolen
- Scrip is validated:
  - By each vendor, on the fly
  - Low computational overhead
  - No network connection
  - No database look up



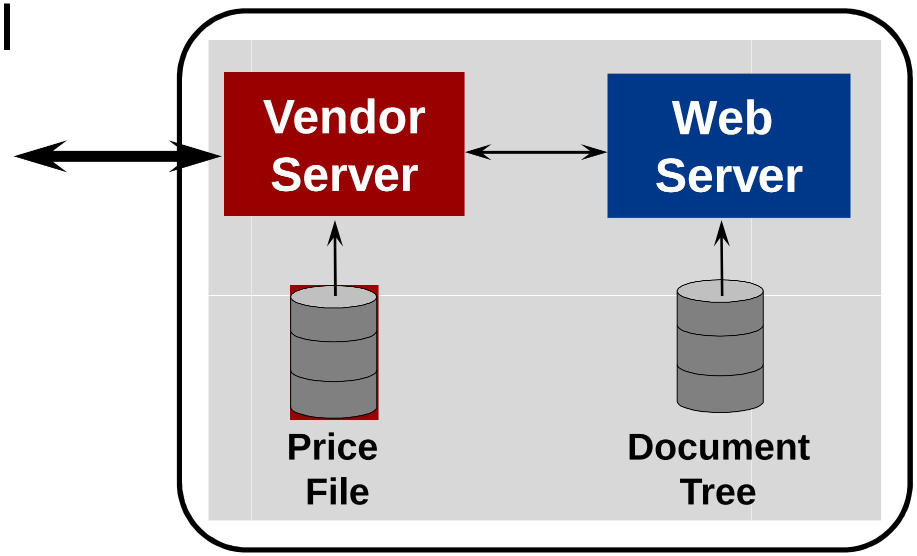
MilliCent Scrip

Vendor	Value	ID#	Cust ID#	Expires	Props	Stamp	Secret

wellsfargo.com / 0.005usd / 0081432 / 101861 / 19961218 {co=us/st=ca} 1d7f4a734b7c02282e48290f04c20

Vendor Server

- Vendor server acts as a proxy for the real Web server
- Vendor server handles all requests:
  - Millicent
  - relay to web-server
- Millicent processing:
  - Validates scrip and generates change
  - Sells subscriptions
  - Handles replays, cash-outs, and refunds



Vendor Site

Major Ideas

- Micropayment systems must be fast and cheap
- They MUST lack features of higher-value payment systems
- Use of hashing instead of cryptography
- Micropayment parties: buyer, seller, broker
- Micromint models minting coins
  - High overhead to prevent counterfeiting
- Fraud is not a serious problem with micropayments

Q&A