Conflict Resolution (OT), Crypto, and Untrusted Cloud Services

COS 418: Distributed Systems
Lecture 17
Michael Freedman

Warning:
This lecture jumps around
But there is some logic + crypto background for blockchain

Today’s Topics

• Conflict resolution
  – Operational Transformation (OT)

• Crypto Introduction
  – Crypto (encryption, digital signatures), hash functions

• Untrusted Cloud Storage (SPORC)
  – OT + crypto + fork* consistency

• Next lecture: Bitcoin and blockchains and consensus, oh my!

Conflict Resolution
Concurrent writes can conflict

- Encountered in many different settings:
  - Peer-to-peer (Bayou)
  - Multi-master: single cluster (Dynamo), wide-area (COPS)
- Potential solutions
  - "Last writer wins"
    - Thomas Write Rule for DBs with timestamp-based concurrency control: Ignore outdated writes
  - Application-specific merge/update: Bayou, Dynamo

General approach: Encode ops as incremental update

- Consider banking (double-entry bookkeeping):
  - Initial: Alice = $50, Bob = $20
  - Alice pays Bob $10
    - Option 1: set Alice to $40, set Bob to $30
    - Option 2: decrement Alice -$10, incremental Bob +$10
      - #2 better, but can't always ensure Alice >= $0
- Works because common mathematical ops are
  - Commutative: A ◦ B == B ◦ A
  - Invertible: A ◦ A⁻¹ == 1

Consider shared word processing

- How do I insert a new word?
  - Send entire doc to server? Not efficient
  - Send update operation!

Consider shared word processing

- How do I insert a new word?
  - Send entire doc to server? Not efficient
  - Send update operation! insert (string, position) = insert("1500s", 166)
  - Warning: Insert (rather than replace) shifted position of all following text
Operations must be commutative

Withdraw $10
$30
Deposit $15
$45
Insert ($"1500s", 166)
$40
Deposit ($"1500s", 166)
$30
Withdraw $10
$55
Delete (1, 0)

[ delete 1 char as pos 0 ]

PROBLEM!
Operational Transformation (OT)

- State of system is $S$, ops $a$ and $b$ performed by concurrently on state $S$
- Different servers can apply concurrent ops in different sequential order
  - Server 1:
    - Receives $a$, applies $a$ to state: $S \odot a$
    - Receives $b$ (which is dependent on $S$, not $S \odot a$)
    - Transforms $b$ across all ops applied since $S$ (namely $a$): $b' = OT(\{a\}, b)$
    - Applies $b'$ to state: $S \odot a \odot b'$
  - Server 2:
    - Receives $b$, applies $b$ to state: $S \odot b$
    - Receives $a$, performs transformation $a' = OT(\{b\}, \{a\})$
    - Applies $a'$ to state: $S \odot b \odot a'$
- Servers 1 and 2 have identical final states: $S \odot a \odot b' == S \odot b \odot a'$
Operational Transformation (OT)

(Used in Google Docs, EtherPad, etc.)

Alice
State: ACE
Ops: ins "ABC" ins "DE" del 2 del 3

Bob
State: ACE
Ops: ins "ABC" ins "DE" del 4 del 2

Intro to crypto in 15 minutes

What is Cryptography?

- From Greek, meaning "secret writing"
- Confidentiality: encrypt data to hide content
- Include “signature” or “message authentication code”
  - Integrity: Message has not been modified
  - Authentication: Identify source of message

Modern encryption:
- Algorithm public, key secret and provides security
- Symmetric (shared secret) or asymmetric (public-private key)
Symmetric (Secret Key) Crypto

- Sender and recipient share common key
  - Main challenge: How to distribute the key?

- Provides dual use:
  - Confidentiality (encryption)
  - Message authentication + integrity (MAC)

- 1000x more computationally efficient than asymmetric

Symmetric Cipher Model

Public-Key Cryptography

- Each party has (public key, private key)

- Alice's public key PK
  - Known by anybody
  - Bob uses PK to encrypt messages to Alice
  - Bob uses PK to verify signatures from Alice

- Alice's private/secret key: sk
  - Known only by Alice
  - Alice uses sk to decrypt ciphertexts sent to her
  - Alice uses sk to generate new signatures on messages

Public-Key Cryptography

- (PK, sk) = generateKey(keysize)

- Encryption API
  - ciphertext = encrypt (message, PK)
  - message = decrypt (ciphertext, sk)

- Digital signatures API
  - Signature = sign (message, sk)
  - isValid = verify (signature, message, PK)
(Simple) RSA Algorithm

- Generating a key:
  - Generate composite \( n = p \cdot q \), where \( p \) and \( q \) are secret primes
  - Pick public exponent \( e \)
  - Solve for secret exponent \( d \) in \( d \cdot e \equiv 1 \pmod{(p - 1)(q - 1)} \)
  - Public key = (\( e, n \)), private key = \( d \)

- Encrypting message \( m \): \( c = m^e \mod n \)
- Decrypting ciphertext \( c \): \( m = c^d \mod n \)

- Security due to cost of factoring large numbers
  - Finding (\( p, q \)) given \( n \) takes \( O(e \log n \log \log n) \) operations
  - \( n \) chosen to be 2048 or 4096 bits long

Cryptographic hash function

( and using them in systems )

Cryptography Hash Functions I

- Take message \( m \) of arbitrary length and produces fixed-size (short) number \( H(m) \)

- One-way function
  - Efficient: Easy to compute \( H(m) \)
  - Hiding property: Hard to find an \( m \), given \( H(m) \)
    - Assumes “\( m \)” has sufficient entropy, not just (“heads”, “tails”)
  - Random: Often assumes for output to “look” random

Cryptography Hash Functions II

- Collisions exist: \(| \) possible inputs \(| >> | \) possible outputs \(|
  … but hard to find

- Collision resistance:
  - Strong resistance: Find any \( m \neq m' \) such that \( H(m) = H(m') \)
  - Weak resistance: Given \( m \), find \( m' \) such that \( H(m) = H(m') \)
  - For 160-bit hash (SHA-1)
    - Finding any collision is birthday paradox: \( 2^{80} \)
    - Finding specific collision requires \( 2^{160} \)
Example use #1: Passwords

- Can’t store passwords in a file that could be read
  - Concerned with insider attacks / break-ins

- Must compare typed passwords to stored passwords
  - Does $H$ (input) == $H$ (password) ?

- Memory cheap: build table of all likely password hashes?
  - Use “salt” to compute $h = H$ (password || salt)
  - Store salt as plaintext in password file, not a secret
  - Then check whether $H$ (input, salt) == $h$

Self-certifying names

- P2P file sharing software (e.g., Limewire)
  - File named by $F_{\text{name}} = H$ (data)
  - Participants verify that $H$ (downloaded) == $F_{\text{name}}$

Hash Pointers

- $h = H$ (data)

Self-certifying names

- BitTorrent
  - Large file split into smaller chunks (~256KB each)
  - Torrent file specifies the name/hash of each chunk
  - Participants verify that $H$ (downloaded) == $C_{\text{name}}$
  - Security relies on getting torrent file from trustworthy source
Hash chains

Creates a “tamper-evident” log of data

If data changes, all subsequent hash pointers change
Otherwise, found a hash collision!

Untrusted Cloud Storage

Operational Transformation
Hash Chains & Digital Signatures
Fork* Linearizability
SPORC: Group Collaboration using Untrusted Cloud Resources

Ariel J. Feldman, William P. Zeller, Michael J. Freedman, Edward W. Felten

OSDI 2010

SPORC goals

Practical cloud apps
- Flexible framework
- Real-time collaboration
- Work offline

Untrusted servers
- Can’t read user data
- Can’t tamper with user data without risking detection
- Clients can recover from tampering

Making servers untrusted

SPORC Server’s limited role:
- Storage
- Ordering msgs
Problem #1
How do you keep clients’ local copies consistent?
(esp. with offline access)

Problem #2:
How do you deal with a malicious server?

Dealing with a malicious server
Digital signatures aren’t enough
Server can equivocate

fork* consistency [LM07]
• Honest server: linearizability
• Malicious server: Alice and Bob detect equivocation after exchanging 2 messages
• Embed hash chain in every msg

Server can still fork the clients, but can’t unfork

System design
System design

Client app

Local state

Committed Pending

fork* consistent

causally consistent

SPORC lib

System design

Client app

Local state

Server

Encrypted state

Encrypt & sign

Committed Pending

SPORC lib

System design

Client app

Local state

Server

Encrypted state

Encrypt & sign

Committed Pending

SPORC lib

System design

Client app

Local state

Server

Encrypted state

Encrypt & sign

Committed Pending

SPORC lib

System design

Client app

Local state

Server

Encrypted state

Encrypt & sign

Committed Pending

SPORC lib

If op\(_k\) depends on op\(_i\), it includes

\[ H(op\(_i\) \| H(op\(_{i-1}\) \| H(op\(_{i-2}\) \ldots)) \]

Possible as ops sequentially ordered by server
**System design**

**Client app**

- Local state
- SPORC lib

**Server Encrypted state**

- Committed
- Pending
- Decrypt & verify

**Recovering from a fork**

- Alice’s history:
  - [ ] [ ] [ ] [ ]

- Bob’s history:
  - [ ] [ ] [ ] [ ]

Can use OT to resolve malicious forks too

**Access control**

**Challenges**

- Server can’t do it — it’s untrusted!
- Preserving causality
- Concurrency makes it harder

**Solutions**

- Ops encrypted with symmetric key shared by clients
- ACL changes are ops too
- Concurrent ACL changes handled with barriers
Summary

• Concurrent operations in eventual-/casual-consistent systems introduce conflicts
  – OT provides general way to merge conflicting ops
  – Newer, more powerful techniques: CRDTs

• Collision resistance in cryptographic hashes can be leveraged to ensure data integrity
  – Used in variety of settings. Key idea in Bitcoin!