Key Value Storage

- Can handle huge volumes of data, e.g., PetaBytes!
  - Store (key, value) tuples

- Simple interface
  - `put(key, value);` // insert/write “value” associated with “key”
  - `value = get(key);` // get/read data associated with “key”

- Used sometimes as a simpler but more scalable “database”

Key Values: Examples

- Amazon:
  - Key: customerID
  - Value: customer profile (e.g., buying history, credit card, ..)

- Facebook, Twitter:
  - Key: UserID
  - Value: user profile (e.g., posting history, photos, friends, …)

- iCloud/iTunes:
  - Key: Movie/song name
  - Value: Movie, Song

Key-Value Storage Systems in Real Life

- Amazon
  - DynamoDB: internal key value store used for Amazon.com (cart)
  - Simple Storage System (S3)

- BigTable/HBase/Hypertable: distributed, scalable data store

- Cassandra: “distributed data management system" (developed by Facebook)

- Memcached: in-memory key-value store for small chunks of arbitrary data (strings, objects)

- BitTorrent distributed file location: peer-to-peer sharing system

...
Key Value Store

- Also called Distributed Hash Tables (DHT)
- Main idea: partition set of key-values across many machines

Challenges

- **Fault Tolerance**: handle machine failures without losing data and without degradation in performance
- **Scalability**:
  - Need to scale to thousands of machines
  - Need to allow easy addition of new machines
- **Consistency**: maintain data consistency in face of node failures and message losses
- **Heterogeneity** (if deployed as peer-to-peer systems):
  - Latency: 1ms to 1000ms
  - Bandwidth: 32Kb/s to 100Mb/s

Key Questions

- **put(key, value)**: where to store a new (key, value) tuple?
- **get(key)**: where is the value associated with a given “key” stored?
- And, do the above while providing
  - Fault Tolerance
  - Scalability
  - Consistency

Directory-Based Architecture

- Directory maintains mapping between keys and machines (nodes) that store the values associated with the keys
- Replicate value on several nodes for fault-tolerance
- Scale by adding nodes, replicating Directory
Directory-Based Architecture

- Directory maintains mapping between keys and machines (nodes) that store the values associated with the keys
- Replicate value on several nodes for fault-tolerance
- Scale by adding nodes, replicating Directory

```
  get(K14)  V14
  K14      V14
  N1        N2
  K5        N50
  K14       N1, N3
  K105      N105
```

Consistency

- Need to make sure that a value is replicated correctly
- How do you know a value has been replicated on every node?
  - Wait for acknowledgements from every node
- What happens if a node fails during replication?
  - Pick another node and try again
- What happens if a node is slow?
  - Slow down the entire put()? Pick another node?
- In general, with multiple replicas
  - Slow puts and fast gets

Large Variety of Consistency Models

- Atomic consistency (linearizability): reads/writes (gets/puts) to replicas appear as if there was a single underlying replica (single system image)
  - Think "one updated at a time"
  - Transactions
- Eventual consistency: given enough time all updates will propagate through the system
  - One of the weakest form of consistency; used by many systems in practice
  - Must eventually converge on single value/key (coherence)
- And many others: causal consistency, sequential consistency, strong consistency, ...

Quorum Consensus

- Improve put() and get() operation performance
- Define a replica set of size N
  - put() waits for acknowledgements from at least W replicas
  - get() waits for responses from at least R replicas
  - W+R > N
- Why does it work?
  - There is at least one node that contains the update
- Why might you use W+R > N+1?
Quorum Consensus Example

- N=3, W=2, R=2
- Replica set for K14: \{N_1, N_3, N_4\}
- Assume put() on N_3 fails

Now, issuing get() to any two nodes out of three will return the answer

What is Computer Security Today?

- Computing in the presence of an adversary!
  - Adversary is the security field’s defining characteristic
- Reliability, robustness, and fault tolerance
  - Dealing with Mother Nature (random failures)
- Security
  - Dealing with actions of a knowledgeable attacker dedicated to causing harm
  - Surviving malice, and not just mischance
- Wherever there is an adversary, there is a computer security problem!

Protection vs. Security

- Protection: mechanisms for controlling access of programs, processes, or users to resources
  - Page table mechanism
  - Round-robin schedule
  - Data encryption
- Security: use of protection mech. to prevent misuse of resources
  - Misuse defined with respect to policy
    » E.g.: prevent exposure of certain sensitive information
    » E.g.: prevent unauthorized modification/deletion of data
  - Need to consider external environment the system operates in
    » Most well-constructed system cannot protect information if user accidentally reveals password – social engineering challenge
Security Requirements

- **Authentication**
  - Ensures that a user is who is claiming to be

- **Data integrity**
  - Ensure that data is not changed from source to destination or after being written on a storage device

- **Confidentiality**
  - Ensures that data is read only by authorized users

- **Non-repudiation**
  - Sender/client can’t later claim didn’t send/write data
  - Receiver/server can’t claim didn’t receive/write data

Securing Communication: Cryptography

- Cryptography: communication in the presence of adversaries

- Studied for thousands of years
  - See the Simon Singh’s *The Code Book* for an excellent, highly readable history

- Central goal: confidentiality
  - How to encode information so that an adversary can’t extract it, but a friend can

- General premise: there is a key, possession of which allows decoding, but without which decoding is infeasible
  - Thus, key must be kept secret and not guessable

Using Symmetric Keys

- Same key for encryption and decryption
- Achieves confidentiality
- Vulnerable to tampering and replay attacks

Symmetric Keys

- Can just XOR plaintext with the key
  - Easy to implement, but easy to break using frequency analysis
  - Unbreakable alternative: XOR with one-time pad
    - Use a different key for each message
Block Ciphers with Symmetric Keys

- More sophisticated (e.g., block cipher) algorithms
  - Works with a block size (e.g., 64 bits)
- Can encrypt blocks separately:
  - Same plaintext ⇒ same ciphertext
- Much better:
  - Add in counter and/or link ciphertext of previous block

Symmetric Key Ciphers - DES & AES

- Data Encryption Standard (DES)
  - Developed by IBM in 1970s, standardized by NBS/NIST
  - 56-bit key (decreased from 64 bits at NSA’s request)
  - Still fairly strong other than brute-forcing the key space
    » But custom hardware can crack a key in < 24 hours
  - Today many financial institutions use Triple DES
    » DES applied 3 times, with 3 keys totaling 168 bits
- Advanced Encryption Standard (AES)
  - Replacement for DES standardized in 2002
  - Key size: 128, 192 or 256 bits
- How fundamentally strong are they?
  - No one knows (no proofs exist)

Authentication in Distributed Systems

- What if identity must be established across network?
  - Need way to prevent exposure of information while still proving identity to remote system
  - Many of the original UNIX tools sent passwords over the wire “in clear text”
    » E.g.: telnet, ftp, yp (yellow pages, for distributed login)
    » Result: Snooping programs widespread
- What do we need? Cannot rely on physical security!
  - Encryption: Privacy, restrict receivers
  - Authentication: Remote Authenticity, restrict senders

Authentication via Secret Key

- Main idea: entity proves identity by decrypting a secret encrypted with its own key
  - K – secret key shared only by A and B
- A can asks B to authenticate itself by decrypting a nonce, i.e., random value, x
  - Avoid replay attacks (attacker impersonating client or server)
- Vulnerable to man-in-the-middle attack
Administrivia

- Midterm 3 Review: Fri 4/20 7:00-10:00PM (VLSB 2050)

- Midterm 3 coming up on Wed 4/25 6:30-8PM
  - All topics up to and including Lecture 23
    » Focus will be on Lectures 17 – 23 and associated readings, and Projects 3
    » But expect 20-30% questions from materials from Lectures 1-16
  - LKS 245, Hearst Field Annex A1, VLSB 2060, Barrows 20, Wurster 102 (see Piazza for your room assignment)
  - Closed book
  - 2 pages hand-written notes both sides

Secure Hash Function

- Hash Function: Short summary of data (message)
  - For instance, \( h_1 = H(M_1) \) is the hash of message \( M_1 \)
    » \( h_1 \) fixed length, despite size of message \( M_1 \)
    » Often, \( h_1 \) is called the “digest” of \( M_1 \)

- Hash function \( H \) is considered secure if
  - It is infeasible to find \( M_2 \) with \( h_1 = H(M_2) \); i.e., can't easily find other message with same digest as given message
  - It is infeasible to locate two messages, \( m_1 \) and \( m_2 \), which “collide”, i.e. for which \( H(m_1) = H(m_2) \)
  - A small change in a message changes many bits of digest/can’t tell anything about message given its hash

Integrity: Cryptographic Hashes

- Basic building block for integrity: cryptographic hashing
  » Associate hash with byte-stream, receiver verifies match
    » Assures data hasn’t been modified, either accidentally – or maliciously

- Approach:
  - Sender computes a secure digest of message \( m \) using \( H(x) \)
    » \( H(x) \) is a publicly known hash function
    » Digest \( d = HMAC(K, m) = H(K \mid H(K \mid m)) \)
    » \( HMAC(K, m) \) is a hash-based message authentication function
  - Send digest \( d \) and message \( m \) to receiver
  - Upon receiving \( m \) and \( d \), receiver uses shared secret key, \( K \), to recompute \( HMAC(K, m) \) and see whether result agrees with \( d \)
Using Hashing for Integrity

Internet

plaintext (m)

HMAC(K,m)

Digest

NO

corrupted msg

m

Encrypted Digest

Digest

HMAC(K,m)

Encrypted Message

Unencrypted Message

Can encrypt m for confidentiality

Standard Cryptographic Hash Functions

- **MD5** (Message Digest version 5)
  - Developed in 1991 (Rivest), produces 128 bit hashes
  - Widely used (RFC 1321)
  - Broken (1996-2008): attacks that find collisions

- **SHA-1** (Secure Hash Algorithm)
  - Developed in 1995 (NSA) as MD5 successor with 160 bit hashes
  - Widely used (SSL/TLS, SSH, PGP, IPSEC)
  - Broken in 2005, government use discontinued in 2010

- **SHA-2** (2001)
  - Family of SHA-224, SHA-256, SHA-384, SHA-512 functions

- HMAC’s are secure even with older “insecure” hash functions

Key Distribution

- How do you get shared secret to both places?
  - For instance: how do you send authenticated, secret mail to someone who you have never met?
  - Must negotiate key over private channel
    » Exchange code book/key cards/memory stick/others

- Could use a third party

Third Party: Authentication Server (Kerberos)

- Notation:
  - $K_{xy}$ is key for talking between x and y
  - $(...)^K$ means encrypt message $(...)$ with the key $K$
  - Clients: A and B, Authentication server S

- Usage:
  - A asks server for key:
    » $A \rightarrow S$: [Hi! I’d like a key for talking between A and B]
    » Not encrypted. Others can find out if A and B are talking
  - Server returns session key encrypted using B’s key
    » $S \rightarrow A$: Message [ Use $K_{ab}$ (This is A! Use $K_{ab}^{K_{sb}}$) $K_{sa}$
    » This allows A to know, “S said use this key”
  - Whenever A wants to talk with B
    » $A \rightarrow B$: Ticket [ This is A! Use $K_{ab}^{K_{sb}}$
    » Now, B knows that $K_{ab}$ is sanctioned by S
Authentication Server Continued [Kerberos]

• Details
  – Both A and B use passwords (shared with key server) to decrypt return from key servers
  – Add in timestamps to limit how long tickets will be used to prevent attacker from replaying messages later
  – Also have to include encrypted checksums (hashed version of message) to prevent malicious user from inserting things into messages/changing messages

• Kerberos is the system that holds all CalNet IDs and passphrases

Asymmetric Encryption (Public Key)

• Idea: use two different keys, one to encrypt (e) and one to decrypt (d)
  – A key pair

• Crucial property: knowing e does not give away d

• Therefore e can be public: everyone knows it!

• If Alice wants to send to Bob, she fetches Bob’s public key (say from Bob’s home page) and encrypts with it
  – Alice can’t decrypt what she’s sending to Bob …
  – … but then, neither can anyone else (except Bob)

Public Key / Asymmetric Encryption

• Sender uses receiver’s public key
  – Advertised to everyone

• Receiver uses complementary private key
  – Must be kept secret

Public Key Encryption Details

• Idea: K\text{public} can be made public, keep K\text{private} private

• Gives message privacy (restricted receiver):
  – Public keys (secure destination points) can be acquired by anyone/used by anyone
  – Only person with private key can decrypt message

• What about authentication?
  – Use combination of private and public key
  – Alice→Bob: [(I’m Alice)^{private} Rest of message]^{public}
  – Provides restricted sender and receiver

• But: how does Alice know that it was Bob who sent her B^{public}? And vice versa…
Public Key Cryptography

- Invented in the 1970s
  - Revolutionized cryptography
  - (Was actually invented earlier by British intelligence)
- How can we construct an encryption/decryption algorithm using a key pair with the public/private properties?
  - Answer: Number Theory
- Most fully developed approach: RSA
  - Rivest / Shamir / Adleman, 1977; RFC 3447
  - Based on modular multiplication of very large integers
  - Very widely used (e.g., ssh, SSL/TLS for https)
- Also mature approach: Eliptic Curve Cryptography (ECC)
  - Based on curves in a Galois-field space
  - Shorter keys and signatures than RSA

Properties of RSA

- Requires generating large, random prime numbers
  - Algorithms exist for quickly finding these (probabilistic!)
- Requires exponentiation of very large numbers
  - Again, fairly fast algorithms exist
- Overall, much slower than symmetric key crypto
  - One general strategy: use public key crypto to exchange a (short) symmetric session key
    » Use that key then with AES or such
- How difficult is recovering d, the private key?
  - Equivalent to finding prime factors of a large number
    » Many have tried - believed to be very hard
    (= brute force only)
    » (Though quantum computers could do so in polynomial time!)

Simple Public Key Authentication

- Each side need only to know the other side's public key
  - No secret key need be shared
- A encrypts a nonce (random num.) x
  - Avoid replay attacks, e.g., attacker impersonating client or server
- B proves it can recover x, generates second nonce y
- A can authenticate itself to B in the same way
- A and B have shared private secrets on which to build private key!
  - We just did secure key distribution!
- Many more details to make this work securely in practice!

Non-Repudiation: RSA Crypto & Signatures

- Suppose Alice has published public key KE
- If she wishes to prove who she is, she can send a message x encrypted with her private key KD (i.e., she sends E(x, KD))
  - Anyone knowing Alice’s public key KE can recover x, verify that Alice must have sent the message
    » It provides a signature
  - Alice can’t deny it: non-repudiation
- Could simply encrypt a hash of the data to sign a document that you wanted to be in clear text
- Note that either of these signature techniques work perfectly well with any data (not just messages)
  - Could sign every datum in a database, for instance
RSA Crypto & Signatures (cont’d)

Alice

I will pay Bob $500

Sign (Encrypt)

Alice’s private key

DFCD3454
BBEA788A

Bob

I will pay Bob $500

Verify (Decrypt)

Alice’s public key

Digital Certificates

- How do you know $K_E$ is Alice’s public key?

- Trusted authority (e.g., Verisign) signs binding between Alice and $K_E$ with its private key $K_{V_{private}}$:
  $C = E(\{Alice, K_E\}, K_{V_{private}})$
  $C$: digital certificate

- Alice: distribute her digital certificate, $C$
- Anyone: use trusted authority’s $K_{V_{public}}$, to extract Alice’s public key from $C$
  $D(C, K_{V_{public}}) = D(E(\{Alice, K_E\}, K_{V_{private}}), K_{V_{public}}) = \{Alice, K_E\}$

Summary of Our Crypto Toolkit

- If we can securely distribute a key, then
  - Symmetric ciphers (e.g., AES) offer fast, presumably strong confidentiality

- Public key cryptography does away with (potentially major) problem of secure key distribution
  - But: not as computationally efficient
    » Often addressed by using public key crypto to exchange a session key

- Digital signature binds the public key to an entity

Putting It All Together - HTTPS

- What happens when you click on https://www.amazon.com?

- https = “Use HTTP over SSL/TLS”
  - SSL = Secure Socket Layer
  - TLS = Transport Layer Security
    » Successor to SSL
  - Provides security layer (authentication, encryption) on top of TCP
    » Fairly transparent to applications
**HTTPS Connection (SSL/TLS) (cont’d)**

- Browser (client) connects via TCP to Amazon’s HTTPS server
- Client sends over list of crypto protocols it supports
- Server picks protocols to use for this session
- Server sends over its certificate
- (all of this is in the clear)

**Inside the Server’s Certificate**

- Name associated with cert (e.g., Amazon)
- Amazon’s RSA public key
- A bunch of auxiliary info (physical address, type of cert, expiration time)
- Name of certificate’s signatory (who signed it)
- A public-key signature of a hash (SHA-256) of all this
  – Constructed using the signatory’s private RSA key, i.e.,
  – Cert = $E(H_{SHA256}(K_{public}^{www.amazon.com}, ...), K_{private})$
    - $K_{public}$: Amazon’s public key
    - $K_{private}$: signatory (certificate authority) private key
    - ...

**Validating Amazon’s Identity**

- How does the browser authenticate certificate signatory?
  – Certificates of several certificate authorities (e.g., Verisign) are hardwired into the browser (or OS)
- If can’t find cert, warn user that site has not been verified
  – And may ask whether to continue
  – Note, can still proceed, just without authentication
- Browser uses public key in signatory’s cert to decrypt signature
  – Compares with its own SHA-256 hash of Amazon’s cert
- Assuming signature matches, now have high confidence it’s indeed Amazon … assuming signatory is trustworthy
  – DigiNotar CA breach (July-Sept 2011): Google, Yahoo!, Mozilla, Tor project, Wordpress, ... (531 total certificates)

**Certificate Validation**

```
Certificate
E(H_{SHA256}(K_{public}^{www.amazon.com}, ...), K_{private}))

E(H_{SHA256}(...), K_{public}) (recall, K_{public} hardwired)

H_{SHA256}(K_{public}^{www.amazon.com}, ...)
```

Validation successful

Can also validate using peer approach: https://www.eff.org/observatory
HTTPS Connection (SSL/TLS) cont’d

- Browser constructs a random session key \( K \) used for data communication
  - Private key for bulk crypto
- Browser encrypts \( K \) using Amazon’s public key
- Browser sends \( E(K, KA_{public}) \) to server
- Browser displays 
- All subsequent comm. encrypted w/ symmetric cipher (e.g., AES128) using key \( K \)
  - E.g., client can authenticate using a password

Summary

- Key-Value Store:
  - Two operations
    - \( \text{put(key, value)} \)
    - \( \text{value = get(key)} \)
  - Challenges
    - Fault Tolerance \( \rightarrow \) replication
    - Scalability \( \rightarrow \) serve get()’s in parallel; replicate/cache hot tuples
    - Consistency \( \rightarrow \) quorum consensus to improve put() performance

- Security:
  - Many more challenges to building secure systems and applications
  - No fixed-point solutions
  - Adversaries constantly change and adapt
  - Defenses must also constantly change and adapt
  - Take CS 161!