University of Surrey, COM3026

Distributed Computing Lecture Notes

Instructors: Dr. Kaitai Liang

Dr. Gregory Chockler

Adrian Coutsoftides

2020 Spring

Contents

1 Moore's law

2 Network partition

3 Marshaling

4 Clock synchronization (L03)

4.1 Berkely Algorithm

4.2 Cristian's algorithm

5 Mutual Exclusion / Lamport (L04)

5.1 Lamport algorithm

5.2 Mutual Exclusion Algorithm

5.3 Leader Election Algorithms

6 Caching (L07)

6.1 Deadlock

Conditions for deadlock

- Mutual Exclusion
- Hold and wait
- Non-preemption
- Circular wait

6.2 UDP Hole punching

7 Symmetric encryption

8 TCP & UDP (L00)

8.1 TCP

8.2 UDP

9 Auth (L08)

9.0.1 Salt

9.0.2 Kerberos

- Developed by MIT
- **Trusted third party**
- Symmetric crypto
- Passwords not sent in clear text
  - Assumes only network is compromised

Kerberos

Users and services authenticate themselves to eachother
To access a service:

- User presents a ticket issues by Kerberos
- Service examines ticket to verify identity of users

Kerberos is trusted third party

- Knows all users and thier Passwords
- Responsible for:
  - Authentication: validating idenity
  - Authorization: deciding whether someone has access to a service
  - Key exchange: sercurely provide both parties with an encryption key

Communication

Alice and Bob both have keys and want to communicate with a service

1. Alice authenticates with Kerberos server
    - Gets **session key** & **sealed envelope**
2. Alice gives Bob the session key (securely)
3. Convinces Bob she also got the session key from Kerberos

□

Authenticate

- Alice asks for permssion to talk to Bob from auth server
- Auth server sends Alice a session key S in a *Ticket* (sealed envelope)

Send key

Alice

- Alice encrypts timestamp with session key
- Sends sealed envelope to Bob

Bob

- Bob decrypts the envelope
    - Gets session key
- Decrypts the timestamp
    - Validates time window
    - Prevent replay attacks

Authenticate recipient

- Bob encrypts Alice's timestamp in the return message
- Alice validates the timestamp
- They communicate using shared **session key** S

9.0.3 Kerboros key usage

Kerberos key usage

- User's password (key) must be used to decode the message from Kerberos everytime a user wants to access a service
- This can be avoided by caching passwords in a file (not a good idea)
- Otherwise, **we can create a temporary password**:
  - Cache temp password
  - Similar to session key for kerberos, to get access to other services
  - Split the Kerberos server into:

$$\text{Auth Server} + \text{Ticket granting server}$$

Definition 1. *TGS + AS = KDC, Kerberos Key Distribution Center*

Auth Server

- Autheticates users, hands out session keys to access TGS
- Before accessing TGS, user requests a ticket to contact TGS

Ticket Granting Server

- Anytime a user wants a service they request it from the TGS
- Reply is encrypted with TGS session key

→ TGS works like a temporary ID.

10 Challenge-handshake

11 Diffie-Hellman (L09)

Secure key distribution is the biggest problem with symmetric cryptography.

Diffie-Hellman Key Exchange

- First algorithm to use public / private keys
- *Not* public key encryption
- Uses a one-way function
  - Based on difficulty of computing discrete logarithms in a finite field compared with ease of calculating exponentiation

→ Allows us to negotiate a secret **common key** without fear of evesdroppers
→ All arithmetic operations are performed in a field of integers modulo some large number

- Both parties agree on a large prime p and a number $\alpha < p$
- Each party generates a public / private key pair
  - Private key for user i: X_i
  - Public key for user i: $Y_i = \alpha^{X_i} \mod p$

Diffie Hellman exponential key exchange

- Alice has a secret key X_A
- Alice has a public key Y_A
- Alice computes $K = Y_B^{X_A} \mod p$

- Bob has a secret key X_B
- Bob has a public key Y_B
- Bob computes $K' = Y_A^{X_B} \mod p$

Definition 2. *Given that $K = K'$, this means K is a **common key** known only to Bob and Alice*

11.1 RSA Public Key Cryptography

- Each user generates 2 keys
  - *Private key* (kept secret)
  - *Public key* (can be shared with anyone)

→ Based on the difficulty of factoring large numbers

Public-key algorithm

- Two related keys:
  - $C = E_{K1}(P)$, $P = D_{K2}(C)$
  - $C' = E_{K2}(P)$, $P = D_{K1}(C')$
  - K_1 is public, K_2 is private
- Examples:
  - RSA & Elliptic Curve Algorithms
  - DSS (digital signature standard)
- Key length:
  - Not every number is a valid key (unlike symmetric crypto)
  - 3072-bit RSA = 256-bit elliptic curve = 128-bit symmetric cipher
  - 15360-bit RSA = 521-bit elliptic curve = 256-bit symmetric cipher

Different keys for encrypting and decrypting

- No need to worry about key distribution
- Share public keys
- Keep private keys secret

→ Alice encrypts her message with Bob's public key, and Bob decrypts it with his private key

11.2 Session keys

Definition 3. *Randomly generated key for one communication session*

- Use public key to send the session key
- Use symmetric algorithm to encrypt data with the session key

Security

- Public key algorithms are almost never used to encrypt messages
  - Much slower, vulnerable to *chosen-plaintext* attacks
  - RSA-2048 approximately 55x slower to encrypt and 2,000x slower to decrypt than AES-256

□

Communication using hybrid crypto-system

- Encrypt message using symmetric algorithm and key K
- Decrypt message using symmetric algorithm and key K

11.3 Hash functions

- Easy to compute in one direction
- Difficult to comput the other way round

$\longrightarrow$ discrete logs or prime factoring
$\longrightarrow$ difficult implies no known shortcuts, requires exhaustive search

Digital Signatures

- Validate
  1. The creator (signer) of the content
  2. The content has not been modified since it was signed
  3. Content itself does not have to be encrypted

Encrypting a message with a private key is the same as signing it
But this is not ideal:

- We don't want to permute or hide the content
- We just want Bob to verify that the content came from Alice
- Public key cryptography is much slower than symmetric encryption
- What if the input was multiple `GB`

Hash Functions

Definition 4. *Cryptographic hash function, also known as a digest*

- Input: arbitary bytes
- Output: fixed-length bit-string
- Properties
  - One-way function
    - Given a hash $H = hash(M)$ it should be difficult to compute M given H
  - Collision resistant
    - Given a hash $H = hash(M)$ it should be difficult to find M' s.t $H = hash(M')$
    - For a hash of length L, a perfect hash would take $2^{L/2}$ attempts
  - Efficient
    - Computing a hash function should be computationally efficient

Popular hash functions

- SHA-2
- SHA-3
  - NIST standard as of 2015
- MD5
  - 128 bits
- Hash functions derived from cyphers
  - Blowfish (used for password hashing in openBSD)
  - 3DES - used for old linux password hashes

□

Digital signatures with hash functions

- Sender
  - Create hash of message
  - Encrypt hash with **private-key** & send it with message
- Recipient
  - Decrypts the encrypted hash using your public key
  - Computes hash of the received message
  - Compare the descrypted hash with the message hash
  - If they're the same the message has not been modified

Message Auth Codes vs. Signatures

- **Message authentication code (MAC)**
- Hash of message encrypted with a symmetric key

- **Digital signature**
- Hash of the message encrypted with the owner's private key
  - Alice encrypts the hash with her private key
  - Bob validates it by decrypting it with her public key & comparing with $hash(M)$
- Provides non-repudiation

Definition 5. *Non-Repudiation means the recipient cannot change the encrypted hash*

□

11.4 Digital Signatures with public key crypto

- Alice generates hash of message
- Alice encrypts has with her **private key**, this is her signature
- Alice sends bob the message and the encrypted hash
- Bob decypts the hash using Alice's public key
- Bob computes the hash of the message sent by Alice
- **If the hashes match, the signature must be valid**

11.4.1 Multiple signers

- Charles
  - Generates hash of message $H(P)$
  - Decrypts Alice's signature with her public key
    - Validates the signature $D_A(S) = H(P)$
  - Decypts Bob's signature with his public key
    - Validates the signature $D_B(S) = H(P)$

11.4.2 Covert & Authenticated Messaging

Combine encryption with digital signatures

Covert Authenticated Messages

Use a **session key**:

- Pick a random key, K to encrypt the message with a symmetric algorithm
- Encrypt K with the public key of each recipient
- For *signing*, **encrypt the hash** of each message with the sender's private key

12 Transactions (L06)

12.1 2-phase commit

12.2 3-phase commit

13 Map-Reduce (Extension slides?)

14 PBFT (L11)

Historical Context

- N generals, f of them are traitors
- Exchange plans by messengers
- - Unreliable
- All loyal generals agree on same plan of action
- Chosen plan must be poposed by loyal general

Computer Science Setting

- A general $\Leftrightarrow$ A program / process / replica
- - Replicas communicate
  - Traitors $\Leftrightarrow$ Failed replicas
- Byzantine army $\Leftrightarrow$ Deterministic replicated service
- - Service has states and some operations
  - Service should cope with failures
  - - State should be consistent across all replicas

Applications

- Replicted file systems
- Backup
- Distributed Servers
- Shared ledgers between banks

□

14.1 Byzantine fault tolerance

- Distributed computing with faulty replicas
  - N replicas
  - f of them faulty
  - Replicas initially start in the same state
- For a given request:
  - Gurantee that all non faulty replicas agree on the next state
  - Provide system consistency even when some replicas may be inconsistent

Properties

- Safety
  - *Agreement*: All non-faulty relpicals agree on the same state
  - *Validity*: The chosen state is valid
- Liveness
  - Some state is eventually agreed upon
  - If a state is chosen all replicas eventually arrive at the same state

□

Paxos vs BFT

Paxos

- Async Network
- Tolerated crash failure
- Guranteed safety but not Liveness
- Protocol may not terminate
- Terminate if the network is synchronous eventually
- **Require at least** $3f + 1$ **replicas to tolerate** f **faulty replicas**

BFT

- Better performance
- Model in BFT is practical
- Adoption in inducstry

□

Byzantine System Model

- Asynchronous distributed system
  - Delay, duplicate or deliver messages out of order
- Byzantine failure model
  - Faulty replicas may behave arbitarily
- Preventing spoofing, relays and corrupted messages
  - Public-key signature: one cannot impersonate the other
  - Message authentication code, collision resistant hash: one cannot tamper the other's messages

Advesary Model

- Can coordinate faulty replicas
- Delay communications but not indefinetly
- Cannot subvert cryptographic techniques employed

Service Properties

- Safety
- Liveness
- Optimal resilliency
  - To tolerate f fulty replicas the system requires $n = 3f + 1$ replicas
  - Can proceed after communicating with $n - f \backsim 2f + 1$ replicas
    * If none of those $2f + 1$ replicas is faulty, good
    * Even if f are faulty, the majority vote $(f + 1)$ ensures safety

Algorithm

- The set of relicas is R where $|R| = 3f + 1$
- Each replica identified by UID $0, \ldots 3f$
- Each replica is deterministic and starts at the same initial state
- **A view is a configuration of replicas**
  - Replica $p = v \mod |R|$ is the primary of view v
  - All other replicas are backups

1. Client sents request to primary
2. Primary validates request and initiates 3-phase protocol to ensure consensus:

$$\text{Pre-Prepare} \longrightarrow \text{Prepare} \longrightarrow \text{Commit}$$

3. Replicas execute request and send it directly back to the client
4. Client accepts result after receiving $f + 1$ identical replies

□

Three phase protocol goals

Ensure safety and liveliness despite asynchronous nature

- Establish total order of execution requests (*Pre-Prepare* + *Prepare*)
- Ensure requests are ordered consistently across views (*Commit*)

Three phases

- Pre-Prepare
  - Achknowledge a unique sequence number for the request
- Prepare
  - The replicas agree on this sequence number
- Commit
  - Establish total order across views

$$\text{Request} \longrightarrow \text{Pre-Prepare} \longrightarrow \text{Prepare} \longrightarrow \text{Commit} \longrightarrow \text{Reply}$$

14.2 Three phase protocol

Definitions

- Request message: m
- Sequence number: n
- Signature: ¥
- View: v
- Primary replica: p
- Digest of message $D(m) \longrightarrow d$

Pre-Prepare

Purpose: acknowledge a unique sequence number for the request

- **Send**
  - The primary assigns the request a sequence number and broadcasts this to all replicas
- A backup will **Accept** a message *iff*:
  - $d, v, n,$ ¥ are valid
  - (v, n) has not been processed before for another digest d

□

Prepare

Purpose: the replicas agree on this sequence number,
$\longrightarrow$ this is after backup i accepts `<Pre-Prepare>` message

- **Send**
  - multicast a `<Prepare>` message acknowledging n, d, i, v
- A replica **Accepts** the message *iff*:
  - $d, v, , n,$ ¥ are valid

□

Prepared

Predicate (m, v, n, i) = `True` *iff* for replica i:

- `<Pre-Prepare>` for m has been received
- $2f + 1$ (including self) is distinct and corresponding `<Prepare>` messages received.

$\square$

Commit

Purpose: establish order across views,
Once (m, v, n, i) = `True` is prepared for replica i:

- **Send**
  - multicast `<Commit>` message to all replicas
- A replica **Accepts** the message *iff*:
  - $d, v, , n,$ ¥ are valid

$\square$

Commited

Predicate (m, v, n, i) = `True` commited *iff* for replica i:

- `prepared`(m, v, n, i) = `True`
- $2f + 1$ (including self) distinct and corresponding valid `<Commit>` message received

Executing Requests

Replica i executes request *iff*:

- `commited`(m, v, n, i) = `True`
- All requests with slower n are already executed

Once executed, the replicas will directly sent `<Reply>` to the client

Everything proceeds as normal if the primary is good, but when the primary is faulty this can cause issues.

14.3 View Change

View Change

If the replica receives requests {1,2,4,5}, it waits to receive request 3 before processing 4 & 5.
Whenever a lot of non-faulty replicas detect that the primary is faulty, they together begin the `view_change` operation.

- If they are stuck, they suspect the primary is faulty
- This fault is detected using timeouts
- This depends in part on the synchrony assumption
- They will then change the view
  - $p \leftarrow (p + 1) \mod |R|$

Initiating the View Change

- Every replica that wants to begin a view change sends a `<View-Change>` message to `Everyone`.
  - Includes current state so that all replicas will know which requests haven't been commited yet (due to faulty primary)
  - List of requests that were prepared
- When the new primary receives $2f + 1$ `<View-Change>` messages, it will begin the view change

□

View-Change & Correctness

- New primary gathers information about which requests need committing
  - This information is included in the `<View-Change>` message
  - All replicas can also compute this sicne they also receive the `<View-Change>` message.
    * Will avoid faulty new primary making the state inconsistent
- New primary sends `<New-View>` to all replicas
- All replicas perform 3 phases on request again

□

View Change fixing missing request

Sequence numbers with missing requests are replaced with a *"no-op / pass"* **operation (null operation).**

State Recomputation

- New primary needs to compute which requests need to be commited again
- Redoing all requests is expensive
- Use checkpoints to speed up progress
  - Every 100 steps, all replicas save their current state into a checkpoint
  - Replicas should agree on the checkpoint as well

Other types of problems include:

- New primary also faulty
  - Use another time-out in the view change
    * When the timeout expires another replica will be chosen as the primary
    * Since there are at most f faulty replicas, **the primary can be consecutively faulty for at most f times**
- Primary might pick disproportionately large sequence number $\backsim 1e^{10}$
  - The sequence number must lie within a certain interval
  - This interval is updated periodically

14.4 Client full protocol

The client may send a request to the primary, but the primary doesnt forward this request to the replicas

Client Full Protocol

- Client sends a request to the primary that they knew
  - The primary may already change, this will be handled
- If they do not receive a reply within a period of time, it broadcasts the request to all replicas

Replica Protocol

- If a replica receives a request from the client but not from the primary, they will forward this request to the primary.
- If they then do not receive a reply from the primary, they begin the `view_change` operation.

14.5 Correctness of View Change

A key proof illustrates why the `view_change` operation preserves **safety**.

Correctness of View Change

Definition 6. *If at any moment the replica has commited a request, this request will* always *be commited in the* `view_change`

- A request is re-commited if they are included in at least one of the `<View-Change>` messages
- A commited request implies there is at least $f + 1$ non-faulty replicas that *prepared* it.

  Proof. ...
  - $\exists\ 2f + 1$ `<View-Change>` messages
  - $\forall m,\ |\,\texttt{prepared}(m)\,| \geq f + 1$
  - If $|R| = 3f + 1$, at least one faulty replica must have prepared m and sent the `<View-Change>` message.

□

The safety lemma is one of the main reasons we use a three pahse protocol instead of a two phase protocol.

- If we only have two phases, we cannot gurantee a request has been commited, it will be prepared by a majority.
- Commited requests will not be recommited, this violates safety

15 Consensus (L10)

15.1 Weak consensus

15.2 Agreement

Gao et al. (2019)

Bibliography

Gao, S., T. Yu, J. Zhu, and W. Cai (2019). T-pbft: An eigentrust-based practical byzantine fault tolerance consensus algorithm. *China Communications 16*(12), 111–123.