بسم الله الرحمن الرحیم

تکنولوژی کامپیوتر

جلسه‌ی پنجم
شبکه

جلسه‌ی گذشته

گولنگِ بیشتر

Map

```go
func main() {
    myMap := make(map[string]int)

    myMap["apple"] = 10
}
```

```go
myMap := map[string]int{
    "apple":  10,
    "banana": 5,
    "orange": 6,
}

delete(myMap, "apple")
```

```go
func main() {
    myMap := map[string]int{
        "apple":  10,
        "banana": 5,
        "orange": 6,
    }

    myMap["apple"] = 25
    val := myMap["apple"]
    fmt.Println(val)

    val2 := myMap["apple2"]
    fmt.Println(val2)

    val2, ok := myMap["apple2"]
    fmt.Println(val2, ok)
}
```

Range

- collection can be a slice, array, map, string, or channel.
- index (or key in the case of maps) is the first return value.
- value is the second return value (the element in slices/arrays, the value in maps, the character (rune) in strings, or the received data from a channel).

```
for index, value := range collection {
    // use index and value
}
```

```
// Only the value
for _, value := range collection {
    // use value only
}
```

```
// Only the index or key
for index := range collection {
    // use index only
}
```

Closure

```go
// An anonymous function assigned to a variable 'increment'.
// The variable 'count' is defined outside the function,
// but is referenced (and modified) by the closure.
increment := func() func() int {
    var count int
    return func() int {
        count++
        return count
    }
}()

fmt.Println(increment()) // Output: 1
fmt.Println(increment()) // Output: 2
fmt.Println(increment()) // Output: 3
```

Closure

```go
func multiplier(factor int) func(int) int {
    return func(x int) int {
        return x * factor
    }
}

func main() {
    mulBy2 := multiplier(2)
    mulBy3 := multiplier(3)

    fmt.Println(mulBy2(10)) // 20
    fmt.Println(mulBy3(10)) // 30
}
```

Closure

- Common Pitfall

```go
func main() {
    var funcs []func()

    for i := 0; i < 3; i++ {
        funcs = append(funcs, func() {
            // This will capture the variable i (not its current value),
            // so all closures end up using i after the loop finishes.
            fmt.Println(i)
        })
    }

    // By the time we call the functions, i is already 3
    for _, f := range funcs {
        f() // Each call prints "3"
    }
}
```

Goroutines

- What Are Goroutines?
  - *Goroutines are lightweight threads managed by the Go runtime. Instead of creating an OS thread directly (which can be expensive in terms of memory and CPU context switches), you spawn a goroutine with the go keyword.*
  - *Under the hood, the Go runtime multiplexes many goroutines onto a small number of OS threads, allowing you to run potentially thousands (or even millions) of concurrent tasks without overwhelming the system.*

Goroutines

```
func doSomething() {
    // some work here
}

// Main function
func main() {
    go doSomething() // spawn a new goroutine
    // ...
}
```

Channels

- What Are Channels?
  - *Channels are typed conduits that allow goroutines to communicate with each other and synchronize without explicit locking.*
- By sending and receiving values on channels, goroutines can coordinate their work, share data safely, and avoid some common concurrency pitfalls.

Channels

```go
// Create a channel of type int
ch := make(chan int)

// Send data to a channel
go func() {
    // blocks until someone receives data
    ch <- 42
}()

// Receive data from a channel
val := <-ch      // blocks until data is available
fmt.Println(val) // 42
```

Channels

- Unbuffered channels
  - *(created via make(chan T))*
  - *block the sender until there is a corresponding receiver.*
- Buffered channels
  - *(created via make(chan T, bufferSize))*
  - *allow sending up to bufferSize elements without blocking. Receivers will still block if there's nothing in the buffer.*

Channels

```
func main() {
    ch := make(chan string)

    go func() {
        ch <- "Hello"
        ch <- "World"
        close(ch)
    }()

    for msg := range ch {
        fmt.Println(msg)
    }
    fmt.Println("Done receiving!")
}
```

```
for {
    msg, ok := <-ch
    fmt.Println(msg, ok)
    if !ok {
        break
    }
}
```

Channels - select

- The select statement in Go allows a goroutine to wait on multiple communication operations (channel sends or receives).
- Whichever communication is ready first is processed, while the others are ignored in that instance.
- If multiple channels are ready, Go picks one at random.

Channels - select

```
func worker(ch1, ch2 <-chan int) {
    for {
        select {
        case x := <-ch1:
            fmt.Println("received from ch1:", x)
        case y := <-ch2:
            fmt.Println("received from ch2:", y)
        default:
            fmt.Println("no one is ready")
            time.Sleep(100 * time.Microsecond)
        }
    }
}
```

Channels - select

```go
func main() {
    ticker := time.NewTicker(1 * time.Second)
    defer ticker.Stop() // ensure we stop the ticker when we're done

    done := make(chan struct{})

    // A goroutine that stops after 5 seconds
    go func() {
        time.Sleep(5 * time.Second)
        done <- struct{}{}
    }()

    // Continuously read from the ticker until done
    for {
        select {
        case t := <-ticker.C:
            fmt.Println("Tick at", t)
        case <-done:
            fmt.Println("Done!")
            return
        }
    }
}
```

Channels - select

```
func process(ctx context.Context, dataChan <-chan int) {
    for {
        select {
        case <-ctx.Done():
            return // exit when cancelled
        case val := <-dataChan:
            // handle val
        }
    }
}
```

context.Context?

- In Go, the context package provides a standardized way to manage deadlines, cancellations, and request-scoped data across multiple API boundaries or goroutines.
- Contexts help you:
  - *Cancel operations when the caller no longer needs the result (e.g., an HTTP client disconnects).*
  - *Set timeouts or deadlines to avoid running tasks indefinitely.*
  - *Carry request-scoped value.*

context.Context?

- An immutable object that is passed between goroutines.
- Base contexts:
  - *context.Background()*
  - *context.TODO()*
- Derived Contexts:
  - *context.WithCancel*
  - *context.WithTimeout*
  - *context.WithDeadline*
  - *context.WithValue*

context.Context

```go
import (
    "context"
    "fmt"
    "time"
)

// doWork simulates a goroutine that periodically does some work
func doWork(ctx context.Context, name string) {
    for {
        select {
        case <-ctx.Done():
            // Context canceled or deadline exceeded
            fmt.Printf("%s: exiting, reason: %v\n", name, ctx.Err())
            return
        default:
            // Simulate work
            fmt.Printf("%s: doing work...\n", name)
            time.Sleep(500 * time.Millisecond)
        }
    }
}
```

context.Context

```go
func main() {
    // Create a base context
    baseCtx := context.Background()

    // Derive a context that automatically cancels after 2 seconds
    ctx, cancel := context.WithTimeout(baseCtx, 2*time.Second)
    defer cancel() // ensure resources are cleaned up

    // Start a worker that checks for cancellation
    go doWork(ctx, "Worker 1")

    // Wait until the context's deadline is reached or cancelled
    <-ctx.Done()
    fmt.Println("Main: context canceled or timed out, reason:", ctx.Err())
}
```

sync.Mutex

- A mutex (short for mutual exclusion) is a concurrency primitive that allows only one goroutine at a time to access a critical section of code (usually shared data).
- In Go, a sync.Mutex provides two methods:
  - *Lock() – acquire the lock (and block if another goroutine holds it).*
  - *Unlock() – release the lock.*
- Only one goroutine can hold the lock at a time. When you need to protect shared resources (e.g., variables, maps, slices) from concurrent access, a mutex ensures that data isn't corrupted by race conditions.

sync.Mutex

```
type AtomicCounter struct {
    counter int
    mu      sync.Mutex
}

func (c *AtomicCounter) Increment() {
    c.mu.Lock() // Acquire the lock
    c.counter++
    c.mu.Unlock() // Release the lock
}

func (c *AtomicCounter) Value() int {
    c.mu.Lock()         // Acquire the lock
    defer c.mu.Unlock() // Release the lock
    return c.counter
}
```

```
func main() {
    c := AtomicCounter{}
    for i := 0; i < 1000; i++ {
        go c.Increment()
    }

    time.Sleep(time.Second)
    fmt.Println("Final counter:", c.Value())
}
```

sync.WaitGroup

- A WaitGroup allows you to wait for a collection of goroutines to finish executing.
- It's essentially a counter:
  - *You increment the counter with Add(n) when you plan to launch n goroutines.*
  - *Each goroutine calls Done() when it finishes (which decrements the counter).*
  - *Wait() blocks until the counter is back to zero (i.e., until all goroutines have called Done()).*

sync.WaitGroup

```go
func worker(id int, wg *sync.WaitGroup) {
    defer wg.Done() // signal we've finished

    fmt.Printf("Worker %d starting\n", id)
    // Do some work...
    fmt.Printf("Worker %d done\n", id)
}

func main() {
    var wg sync.WaitGroup

    const numWorkers = 5
    wg.Add(numWorkers) // we have 5 workers
    for i := 1; i <= numWorkers; i++ {
        go worker(i, &wg)
    }

    wg.Wait() // wait for all workers to finish
    fmt.Println("All workers completed")
}
```

جلسه‌ی جدید

گولنگِ بیشتر - شبکه

ماژول در گولنگ

دانلود پکیج و کار کردن با چندتا پکیج

JSON

- معرفی struct tag
- با مثال json

شبکه

لایه‌های شبکه در مدل TCP/IP

- Link Layer
  - *Describes protocols for connecting hosts to the local network for data transfer.*
  - *Examples: Ethernet, Wi-Fi, ARP (Address Resolution Protocol).*

لایه‌های شبکه در مدل TCP/IP

- Internet Layer
  - *Handles the movement of packets around the network via IP addressing and routing.*
  - *Protocols: IP (Internet Protocol)*

لایه‌های شبکه در مدل TCP/IP

- Transport Layer
  - *Manages end-to-end communication*
  - *and reliability or unreliability (TCP vs. UDP).*
  - *Protocols: TCP, UDP, (ICMP).*

لایه‌های شبکه در مدل TCP/IP

- Application Layer
  - *Provides application services to end users.*
  - *Protocols: HTTP, FTP, SMTP, DNS, SSH, etc.*

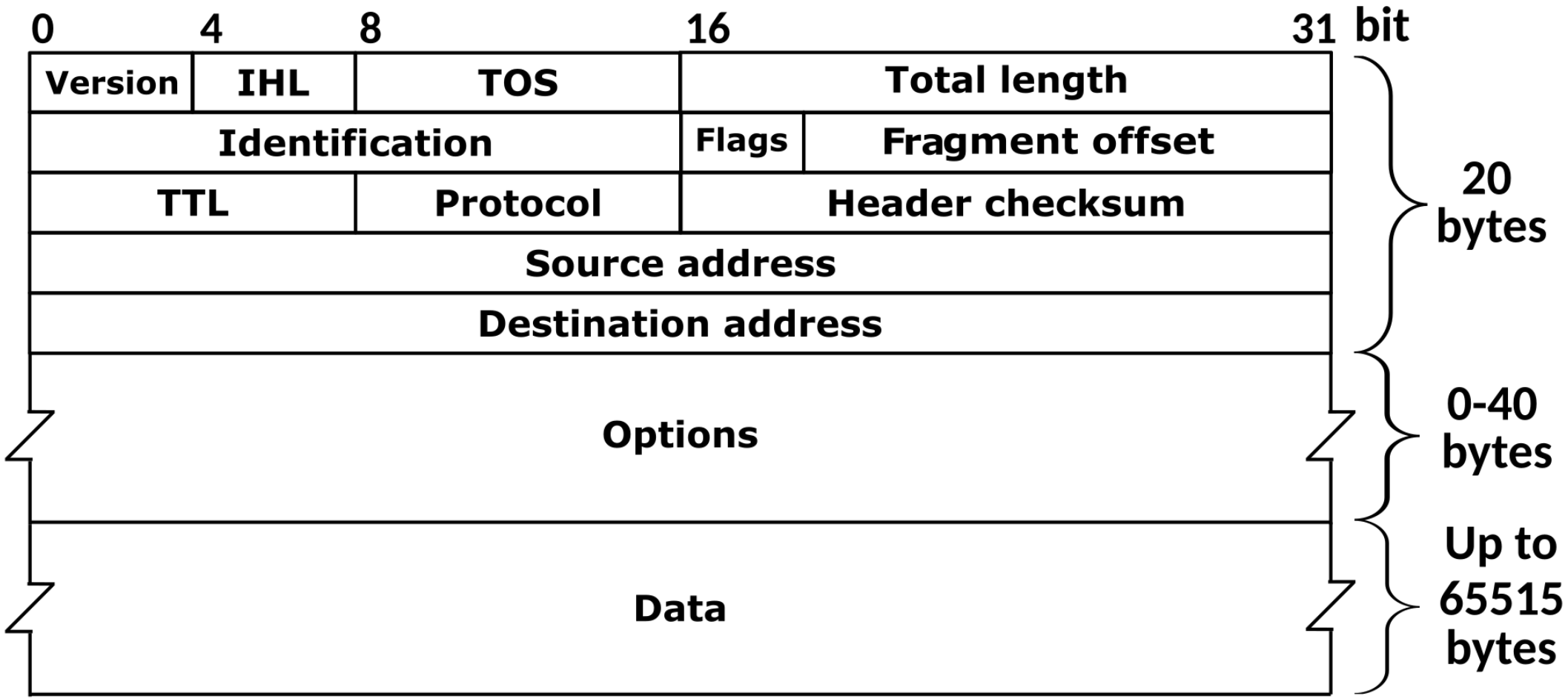
IP

- هر موجودی به به شبکه وصل است، یک (یا چند) آدرس (که به آن IP) می‌گوییم دارد.
- در IPv4، این آدرس ۳۲ بیت است.
- برای نمایش از ۴ عدد بین ۰ تا ۲۵۵ که با نقطه جدا شدند استفاده می‌شود.
  - *127.0.0.1*
  - *204.18.190.221*

IP

- این ipها، بعضی‌هاش به طور خاص استفاده می‌شه.
  - *مثلا: 127.0.0.1 برای آی‌پی مشخص کردن local host، همین سروری که توش هستیم.*
- CIDR و ip subnet، آدرس‌های IP رو به نوعی کلاس‌بندی می‌کنیم.
  - *مثلا: 127.0.0.1/8 برای localhostها رزرو شده*

IP

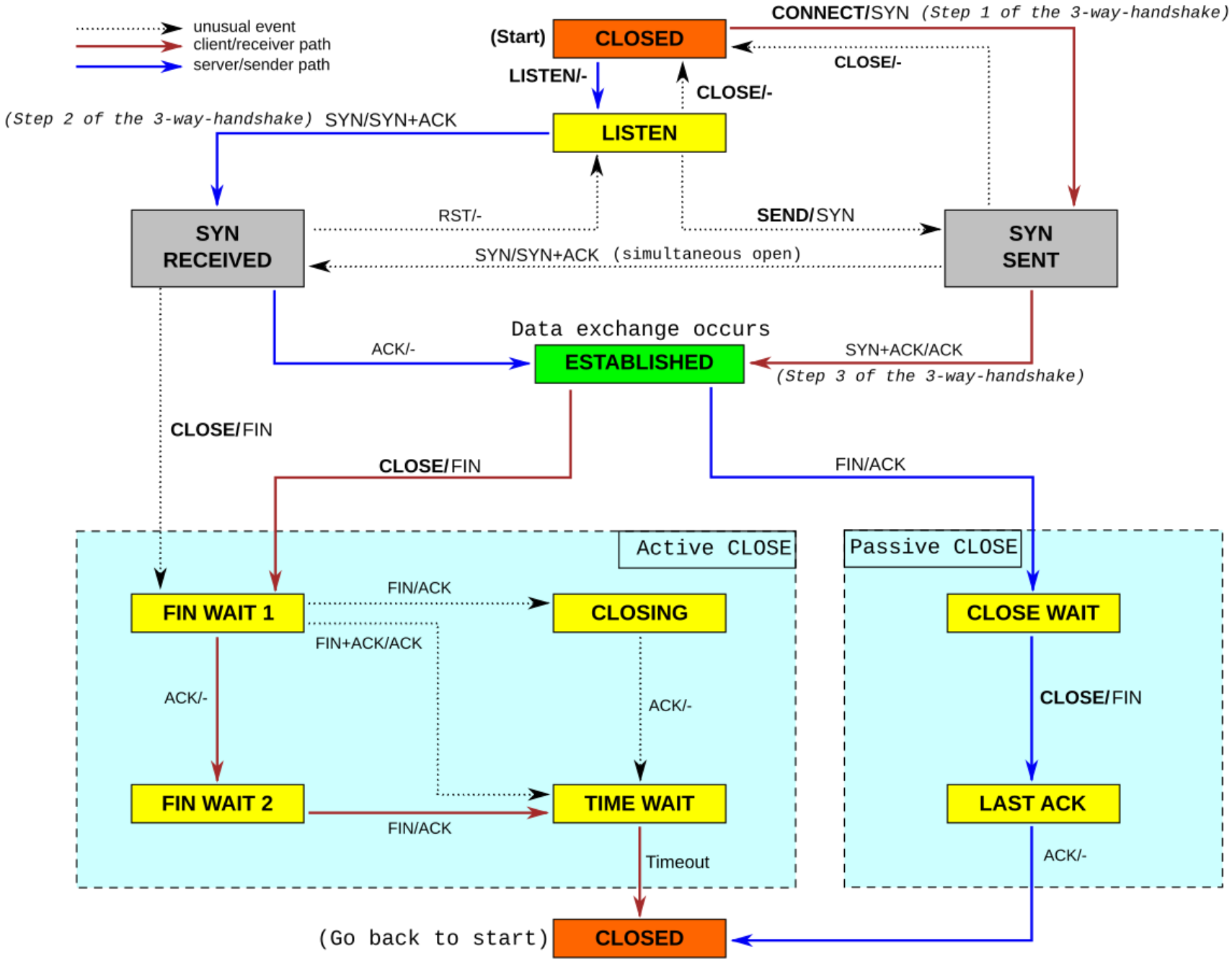

0
4
8
16
31
bit
Version
IHL
TOS
Total length
Identification
Flags
Fragment offset
TTL
Protocol
Header checksum
Source address
Destination address
Options
Data
20 bytes
0-40 bytes
Up to 65515 bytes

UDP

UDP header format[7]

Offset	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	*Source Port*																*Destination Port*															
4	32	*Length*																*Checksum*															
8	64	*Data*																															
12	96																																
⋮	⋮																																

TCP


unusual event
client/receiver path
server/sender path
(Start)
CLOSED
CONNECT/SYN (Step 1 of the 3-way-handshake)
CLOSE/-
LISTEN/-
CLOSE/-
LISTEN
(Step 2 of the 3-way-handshake) SYN/SYN+ACK
SYN RECEIVED
RST/-
SEND/SYN
SYN SENT
SYN/SYN+ACK (simultaneous open)
Data exchange occurs
ACK/-
ESTABLISHED
SYN+ACK/ACK
(Step 3 of the 3-way-handshake)
CLOSE/FIN
CLOSE/FIN
FIN/ACK
Active CLOSE
Passive CLOSE
FIN WAIT 1
FIN/ACK
CLOSING
CLOSE WAIT
FIN+ACK/ACK
ACK/-
ACK/-
CLOSE/FIN
FIN WAIT 2
TIME WAIT
LAST ACK
FIN/ACK
Timeout
ACK/-
(Go back to start)
CLOSED

TCP

TCP header format[17]

Offset	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Source Port																Destination Port															
4	32	Sequence Number																															
8	64	Acknowledgement Number (meaningful when ACK bit set)																															
12	96	Data Offset				Reserved				CWR	ECE	URG	ACK	PSH	RST	SYN	FIN	Window															
16	128	Checksum																Urgent Pointer (meaningful when URG bit set)[18]															
20	160	(Options) If present, Data Offset will be greater than 5. Padded with zeroes to a multiple of 32 bits, since Data Offset counts words of 4 octets.																															
⋮	⋮																																
56	448																																
60	480	Data																															
64	512																																
⋮	⋮																																

TCP

- Reliable Data Transfer: Lost or corrupt segments are detected and retransmitted.
- Ordered Delivery: Sequence numbers ensure data is reassembled in the correct order.
- Flow Control: The receiving end can tell the sender how much data it can handle at once.
- Congestion Control: TCP tries to sense network congestion and adjust the sending rate, helping to avoid overwhelming the network.
- Connection-Oriented: The handshake before data transfer ensures both ends agree on parameters (sequence numbers, MSS, etc.).