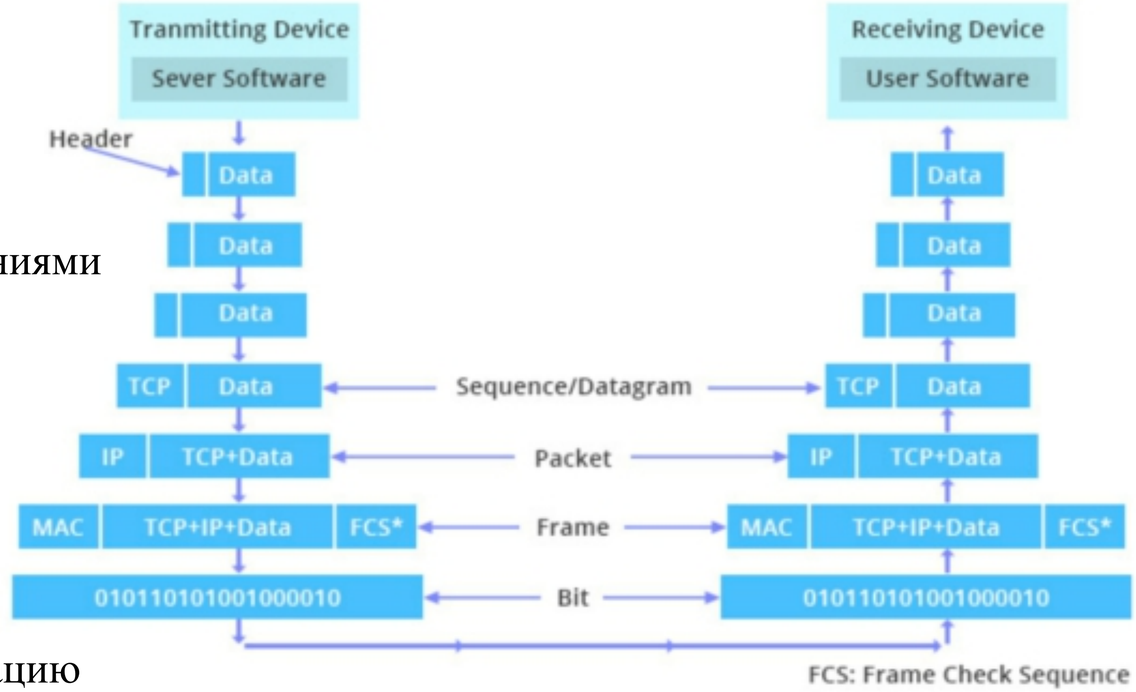
Модели TCP/IP и OSI

Модель TCP/IP

1) **Прикладной уровень** – протоколы , используемые приложениями для обмена данными через соединения , установленные низлежащими протоколами
   · HTTP, FTP, SMTP, DHCP, etc
2) **Транспортный уровень** – отвечает за коммуникацию между приложениями , разбивает данные на сегменты
   · TCP, UDP, QUIC
3) **Межсетевой уровень** – отвечает за адресацию и маршрутизацию пакетов данных между сетями
   · IP, ICMP, etc
4) **Канальный** – отвечает за физическую передачу данных в локальной сети
   · Ethernet , PPP, WIFI



Модель OSI		Модель TCP/IP	
Прикладной уровень (application layer)	7	4	Прикладной уровень (application layer)
Уровень представления (presentation layer)	6		
Сеансовый уровень (session layer)	5		
Транспортный уровень (transport layer)	4	3	Транспортный уровень (transport layer)
Сетевой уровень (network layer)	3	2	Межсетевой уровень (internet layer)
Канальный уровень (data link layer)	2	1	Канальный уровень (link layer)
Физический уровень (physical layer)	1		

Модель OSI

Данные	**Прикладной** (доступ к сетевым службам)	Осуществляет взаимодействие между пользователем и сетью. Взаимодействует с приложениями на стороне клиента.	HTTP, FTP, Telnet, SSH, SNMP
Данные	**Представления** (представление и кодирование данных)	Осуществляет преобразование данных в нужную форму, шифрование/кодирование, сжатие.	MIME, SSL
Данные	**Сеансовый** (управление сеансом связи)	Управляет созданием/поддержанием/завершением сеанса связи.	L2TP, RTCP
Блоки	**Транспортный** (безопасное и надежное соединение точка-точка)	Предназначен для доставки данных без ошибок, потерь и дублирования в той последовательности, как они были переданы. Выполняет сквозной контроль передачи данных от отправителя до получателя.	TCP, UDP
Пакеты	**Сетевой** определение пути и IP (логическая адресация)	Его основными задачами являются маршрутизация – определение оптимального пути передачи данных, логическая адресация узлов. Кроме того, на этот уровень могут быть возложены задачи по поиску неполадок в сети (протокол ICMP). Сетевой уровень работает с пакетами.	IP, ICMP, IGMP, BGP, OSPF
Кадры	**Канальный** MAC и LLC (физическая адресация)	Отвечает за доступ к среде передачи, исправление ошибок, надежную передачу данных. *На приеме* полученные с физического уровня данные упаковываются в кадры после чего проверяется их целостность. Если ошибок нет, то данные передаются на сетевой уровень. Если ошибки есть, то кадр отбрасывается и формируется запрос на повторную передачу. Канальный уровень подразделяется на два подуровня: **MAC (Media Access Control)** и **LLC (Locical Link Control)**. MAC регулирует доступ к разделяемой физической среде. LLC обеспечивает обслуживание сетевого уровня. На канальном уровне работают коммутаторы.	IEEE 802.3, IEEE 802.11, PPP, DHCP, ARP
Биты	**Физический** (кабель, сигналы, бинарная передача данных)	Определяет вид среды передачи данных, физические и электрические характеристики интерфейсов, вид сигнала. Этот уровень имеет дело с битами информации.	IEEE 802.11, ISDN

HTTP

HTTP

Hyper Text Transfer Protocol.
Протокол прикладного уровня , разработанный в рамках модели TCP/IP.

HTTP семантика

Метод запроса (Method)

HTTP Method	Description	Request has body	Response has body	Safe	Idempotent	Cacheable
GET	Requests the representation of a resource. The primary information retrieval mechanism.	no	yes	yes	yes	yes
HEAD	Retrieves only metadata without the actual content associated with the resource. Can be used for testing for accessibility or recent modifications.	no	no	yes	yes	yes
POST	Requests server processing of an attached payload according to its own semantics. Can be used to submit a form, post a message, or add items to a database...	yes	yes	no	no	yes
PUT	Replaces (or creates) the representation of a resource. Mostly used to update data or change the status/value of a resource.	yes	yes	no	yes	no
DELETE	Requests server removal of a specified resource. It is up to the server to archive or actually delete information.	no	yes	no	yes	no
OPTIONS	Learn server capabilities and allowed methods and options for a given resource. Can be used to test and discover what the server allows.	yes/no	yes	yes	yes	no
CONNECT	Requests that the recipient (proxy) initiates a tunnel to the destination server. Used to establish virtual connections and secure/encrypt communications.	yes	yes	no	no	no
TRACE	Remote loop-back echo of the request received by the server. Used for testing/diagnostic/debug purposes.	no	yes	yes	yes	no

HTTP семантика

Заголовок запроса /ответа (Header)

Accept, Connection, Authorization, etc
Возможно задавать собственные заголовки
Список общеизвестных заголовков :
https://en.wikipedia.org/wiki/List_of_HTTP_header_fields#Response_fields

HTTP семантика

Код статуса ответа (Status code)

HTTP Status Codes

javaconceptoftheday.com

1xx : Informational Purpose	
100	Continue
101	Switching Protocols
102	Processing
103	Early Hints

2xx : Success	
200	Ok
201	Created
202	Accepted
203	Non-Authoritative Information
204	No Content
205	Reset Content
206	Partial Content
207	Multi Status
208	Already Reported
226	IM Used

3xx : Redirection	
300	Multiple Choices
301	Moved Permanently
302	Found
303	See Other
304	Not Modified
305	Use Proxy
306	No Longer Used
307	Temporary Redirect
308	Moved Permanently

4xx : Client Errors	
400	Bad Request
401	Unauthorized
402	Payment Required
403	Forbidden
404	Not Found
405	Method Not Allowed
406	Not Acceptable
407	Proxy Authentication Is Required
408	Request Time Out
409	Conflict
410	Gone
411	Length Required
412	Precondition Failed
413	Payload Too Large
414	URI Too Long
415	Unsupported Media Type
416	Range Not Satisfiable
417	Expectation Failed
421	Misdirect Request
422	Unprocessable Entity
423	Locked
424	Failed Dependency
425	Too Early
426	Upgrade Required
428	Precondition Required
429	Too Many Requests
431	Request Header Fields Too Large
451	Unavailable For Legal Reasons

5xx : Server Errors	
500	Internal Server Error
501	Not Implemented
502	Bad Gateway
503	Service Unavailable
504	Gateway Timeout
505	HTTP Version Not Supported
507	Insufficient Storage
508	Loop Detected
510	Not Extended
511	Network Authentication Required

HTTP/1

Как транспортный протокол использует TCP

HTTP

В Go для работы с HTTP есть пакет “http”. Он предоставляет реализацию HTTP-клиента и сервера.

HTTP client

HTTP клиент

Пример создания HTTP-клиента:

```
client := &http.Client{
   CheckRedirect: redirectPolicyFunc,
}

resp, err := client.Get("http://example.com")
// ...

req, err := http.NewRequest("GET", "http://example.com", nil)
// ...
req.Header.Add("If-None-Match", `W/"wyzzy"`)
resp, err := client.Do(req)
// ...
```

HTTP клиент

Существует возможность более точной конфигурации клиента:

```
tr := &http.Transport{
  MaxIdleConns:       10,
  IdleConnTimeout:    30 * time.Second,
  DisableCompression: true,
}
client := &http.Client{Transport: tr}
resp, err := client.Get("https://example.com")
```

HTTP server

HTTP сервер

HTTP-сервер запускается запускается с помощью функции http.ListenAndServe:

```
http.Handle("/foo", fooHandler)

http.HandleFunc("/bar", func(w http.ResponseWriter, r *http.Request) {
  fmt.Fprintf(w, "Hello, %q", html.EscapeString(r.URL.Path))
})

log.Fatal(http.ListenAndServe(":8080", nil))
```

HTTP сервер

Либо вызовом метода ListenAndServe объекта http.Server:

```
s := &http.Server{
   Addr:           ":8080",
   Handler:        myHandler,
   ReadTimeout:    10 * time.Second,
   WriteTimeout:   10 * time.Second,
   MaxHeaderBytes: 1 << 20,
}
log.Fatal(s.ListenAndServe())
```

HTTP сервер

HTTP-сервер имеет мультиплексор (mux, роутер) для распределения запросов на разные ендпоинты соответствующим функциям-обработчикам. По умолчанию используется дефолтовый мультиплексор:

```
http.Handle("/foo", fooHandler)

http.HandleFunc("/bar", func(w http.ResponseWriter, r *http.Request) {
  fmt.Fprintf(w, "Hello, %q", html.EscapeString(r.URL.Path))
})
```

HTTP сервер

Можно создать свой, для более удобной организации обработчиков:

```
mux := http.NewServeMux()
mux.HandleFunc("/task/{id}/status/", func(w http.ResponseWriter, r *http.Request) {
        id := r.PathValue("id")
        fmt.Fprintf(w, "handling task status with id=%v\n", id)
})
mux.HandleFunc("/task/0/{action}/", func(w http.ResponseWriter, r *http.Request) {
        action := r.PathValue("action")
        fmt.Fprintf(w, "handling task 0 with action=%v\n", action)
})
```

HTTP сервер

Так же существует возможность работы со статическими файлами:

```
http.FileServer(http.Dir(“static"))
```

```
http.ServeFile(w, r, "static/index.html")
```

HTTP сервер

Возможно добавлять свои промежуточные обработчики (middleware):

```
func loggingMiddleware(next http.Handler) http.Handler {
        return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
            log.Printf("Request received: %s %s", r.Method, r.URL.Path)
            next.ServeHTTP(w, r)
        })
}
```

HTTP сервер

Для более удобной работы с маршрутизацией запросов используются библиотеки:

- https://github.com/gorilla/mux
- https://github.com/gin-gonic/gin

HTTP сервер

Каждый запрос обрабатывается в отдельной горутине.
Пакет “http” использует netpoller для обработки входящих запросов.
Подробнее про его работу можно почитать тут:
https://goperf.dev/02-networking/networking-internals/#internals-of-the-net-package

Тестирование HTTP запросов

Тестирование HTTP запросов

Для тестирования исходящих запросов и обработчиков входящих запросов есть пакет “httptest”.

Тестирование HTTP запросов

Тестирование исходящих запросов:

```
server := httptest.NewServer(http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
   w.WriteHeader(…)
   w.Write([]byte(…))
  }))
defer server.Close()
```

В тесте создаётся тестовый сервер и далее клиент отправляет ему запросы.

Тестирование HTTP запросов

Тестирование обработчиков запросов:

```
handler := func(w http.ResponseWriter, r *http.Request) {
   io.WriteString(w, "<html><body>Hello World!</body></html>")
}

req := httptest.NewRequest("GET", "http://example.com/foo", nil)
w := httptest.NewRecorder()
handler(w, req)

resp := w.Result()
body, _ := io.ReadAll(resp.Body)
```

Создаются тестовые объекты http.Request и http.ResponseWriter.
Далее вызывается обработчик и проверяются результаты.

Шаблонизация

Шаблонизация

Для генерирования файлов html из шаблонов существует пакет “html/templates”.

```
http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {

  data := ViewData{
    Title: "World Cup",
    Message: "FIFA will never regret it",
  }
  tmpl, _ := template.ParseFiles("templates/index.html")
  tmpl.Execute(w, data)
})
```

Также существует пакет “html/templates” для генерирования текстовых файлов html из шаблонов.

Трассировка HTTP запроса

Трассировка HTTP запроса

С помощью пакета “httptrace” можно отслеживать события HTTP-запроса и получать о них информацию:

```
req, _ := http.NewRequest("GET", "http://example.com", nil)
   trace := &httptrace.ClientTrace{
      GotConn: func(connInfo httptrace.GotConnInfo) {
         fmt.Printf("Got Conn: %+v\n", connInfo)
      },
      DNSDone: func(dnsInfo httptrace.DNSDoneInfo) {
         fmt.Printf("DNS Info: %+v\n", dnsInfo)
      },
   }
   req = req.WithContext(httptrace.WithClientTrace(req.Context(), trace))
```