Osi – open systems interconnection

ფიზიკური ფენა (Physical Layer) არის ყველაზე დაბალი ფენა, რომელიც პასუხისმგებელია მონაცემთა ფიზიკურ გადაცემაზე. იგი

განსაზღვრავს, თუ როგორ გადაიცემა ბიტები (0 და 1) ერთი მოწყობილობიდან მეორეზე კაბელების, სიგნალების და ინტერფეისების მეშვეობით.

მაგალითად, ელექტრონული სიგნალები, რადიო ტალღები ან სინათლის პულსები.

მონაცემთა ბმული ფენა (Data Link Layer) - ეს ფენა უზრუნველყოფს მონაცემთა საიმედო გადაცემას ფიზიკურ ფენაზე. იგი დაყოფილია ორ ქვეფენად:

MAC (Media Access Control) და LLC (Logical Link Control). MAC ქვეფენა აკონტროლებს, თუ როგორ იღებენ მოწყობილობები წვდომას მედიაზე

(მაგ., Ethernet), ხოლო LLC ქვეფენა აკონტროლებს შეცდომების გამოვლენას და შესწორებას. ამ ფენაზე მუშაობს, მაგალითად, Ethernet და Wi-Fi.

ქსელური ფენა (Network Layer) - ეს ფენა პასუხისმგებელია მონაცემთა პაკეტების მარშრუტიზაციაზე სხვადასხვა ქსელებს შორის. იგი განსაზღვრავს,

თუ როგორ მიაღწევს მონაცემები წყაროდან დანიშნულების ადგილამდე, IP მისამართების გამოყენებით. ამ ფენაზე მუშაობს პროტოკოლები,

როგორიცაა IP (Internet Protocol) და როუტერები.

ტრანსპორტული ფენა (Transport Layer) ეს ფენა უზრუნველყოფს მონაცემთა საიმედო გადაცემას წყაროსა და მიმღებს შორის. იგი აკონტროლებს

მონაცემთა ნაკადის სიჩქარეს, შეცდომების გამოვლენას და გამოსწორებას. ამ ფენაზე გამოიყენება პროტოკოლები, როგორიცაა TCP (Transmission

Control Protocol) და UDP (User Datagram Protocol).

სესიის ფენა (Session Layer) ეს ფენა აკონტროლებს კომუნიკაციის სესიებს მოწყობილობებს შორის. იგი პასუხისმგებელია სესიების დაწყებაზე,

მართვაზე და დასრულებაზე, ასევე დიალოგის სინქრონიზაციაზე. მაგალითად, იგი უზრუნველყოფს, რომ მონაცემები გადაიცემა სწორად და სესია არ

გაწყდება მოულოდნელად.

წარმოდგენის ფენა (Presentation Layer) ეს ფენა პასუხისმგებელია მონაცემთა ფორმატირებაზე, დაშიფვრაზე და დეშიფრაზე. იგი უზრუნველყოფს,

რომ მონაცემები იყოს გასაგები გამომგზავნისა და მიმღებისთვის. მაგალითად, იგი ასრულებს მონაცემთა კომპრესიას, დაშიფვრას და ASCII-ში

გადაყვანას.

აპლიკაციური ფენა (Application Layer) ეს არის ყველაზე მაღალი ფენა, რომელიც უშუალო კავშირს უზრუნველყოფს მომხმარებელსა და ქსელს

შორის. იგი უზრუნველყოფს სერვისებს, როგორიცაა ელექტრონული ფოსტა, ფაილების გადაცემა, ვებ-გვერდების ჩვენება და სხვა. ამ ფენაზე მუშაობს

პროტოკოლები, როგორიცაა HTTP, FTP, SMTP და DNS.

სტანდარტი როგორ ილაპარაკონ 2მა კონპიუტერმა ერთმანეთში

სანამ პროგრამის კოდს დაწერ, **აუცილებელია გქონდეს გეგმა — ანუ არქიტექტურა**, თუ **როგორ იქნება მოწყობილი შენი აპლიკაცია**.

**რა განსხვავებაა "აპლიკაციის არქიტექტურასა" და "ქსელის არქიტექტურას" შორის?**

* **ქსელის არქიტექტურა** (მაგ. ინტერნეტის 5-ფენიანი მოდელი) — უკვე მზა სისტემაა, რომლის შეცვლაც შენ არ შეგიძლია.  
  ➤ ის მხოლოდ გაძლევს რაღაც სერვისებს, როგორებიცაა მონაცემის გადაცემა, IP მისამართები, და ა.შ.
* **აპლიკაციის არქიტექტურა** — ამას შენ ქმნი.  
  ➤ შენ წყვეტ, **როგორ იმუშავებს შენი აპლიკაცია სხვადასხვა კომპიუტერზე ან მოწყობილობაზე**.
* .

**🏗️ აპლიკაციის არქიტექტურის ტიპები:**

როდესაც გადაწყვეტ, როგორ გააწყო შენი აპლიკაცია, არჩევა ხშირად ერთ-ერთ ასეთ მოდელს შორის უნდა გააკეთო:

1. **Client-Server არქიტექტურა**  
   👉 არსებობს **ერთი ცენტრალური სერვერი** და **რამდენიმე კლიენტი**, რომლებიც უკავშირდებიან ამ სერვერს  
   მაგ: Facebook, Gmail — შენ ხარ კლიენტი, და უკავშირდები სერვერს, სადაც შენს ინფორმაციას ინახავს
2. **Peer-to-Peer (P2P) არქიტექტურა**  
   👉 ყველა კომპიუტერი ერთმანეთთან არის დაკავშირებული, და ყველა ერთდროულად კლიენტიც არის და სერვერიც  
   მაგ: Torrent-ები, სადაც ფაილს სხვა ადამიანების კომპიუტერებიდან იღებ და შენც აწვდი სხვებს

**🌐 Client-Server არქიტექტურა**

ეს მოდელი არის ერთ-ერთი ყველაზე გავრცელებული და მარტივი პრინციპი, რომელსაც ინტერნეტში ძალიან ბევრი აპლიკაცია იყენებს.

**🧑‍💻 რა ხდება ამ მოდელში?**

* არსებობს **ერთი ძლიერი კომპიუტერი**, რომელსაც ეძახიან **სერვერს (Server)**.  
  ➤ ის **ყოველთვის ჩართულია** და მზად არის მოემსახუროს მოთხოვნებს.
* არიან **მომხმარებლები**, ანუ ჩვეულებრივი კომპიუტერები, ლეპტოპები ან ტელეფონები — მათ ეძახიან **კლიენტებს (Clients)**.  
  ➤ კლიენტები აგზავნიან მოთხოვნებს სერვერზე და სერვერი პასუხობს.

**🧵 მაგალითი:**

📄 წარმოიდგინე, რომ შენ ბრაუზერში (Chrome, Firefox) ეძებ www.example.com.

* შენი კომპიუტერი (კლიენტი) აგზავნის მოთხოვნას იმ ვებსერვერზე, სადაც example.com-ის საიტია გაშვებული.
* სერვერი უბრუნებს ვებსაიტის შინაარსს — სურათებს, ტექსტს და ყველაფერს.
* შენ მხოლოდ სერვერთან გაქვს კავშირი, სხვა მომხმარებლებთან არა.

**🧭 დამატებითი მახასიათებლები:**

* **სერვერს აქვს ფიქსირებული მისამართი**, ანუ **IP მისამართი**, რომელსაც ყველა კლიენტი იყენებს მასთან დასაკავშირებლად.
* **კლიენტები ერთმანეთთან პირდაპირ არ ურთიერთობენ**.  
  მაგ: თუ შენ და მეგობარი ორივე ხართ Facebook-ზე, თქვენი კავშირი ჯერ Facebook-ის სერვერზე გადის.

**📬 რომელ აპლიკაციებში გამოიყენება?**

ყველაზე ცნობილი მაგალითებია:

* 🌐 **ვებსაიტები** (Web – ბრაუზერით მუშაობა)
* 📂 **FTP** – ფაილების გადაგზავნა
* 💻 **Telnet** – დისტანციური კავშირი სერვერთან
* ✉️ **ელფოსტა** – წერილები მიდის სერვერზე და მერე შენ იღებ

## 🏛 Client-Server vs. P2P არქიტექტურა – მარტივად

### ☁️ ****რა პრობლემა აქვს მხოლოდ ერთ სერვერზე დაყრდნობას?****

თუ ბევრი ადამიანი ერთდროულად იყენებს აპლიკაციას (მაგ. Facebook, Amazon), **ერთი სერვერი ვეღარ ასწრებს ყველას მომსახურებას**.

🔹 ამიტომ დიდ კომპანიებს (Google, Facebook, Amazon) აქვთ **დატა ცენტრები** – უზარმაზარი შენობები, სადაც ათასობით სერვერია მოთავსებული.

* Google-ს, მაგალითად, მსოფლიოში 30-50 მონაცემთა ცენტრი აქვს.
* ეს სერვერები საჭიროებს ელექტროენერგიას, ტექნიკურ მომსახურებას და ინტერნეტთან ძლიერ კავშირს – რაც დიდ ფულად ხარჯებთანაა დაკავშირებული.

### 🤝 ****P2P (Peer-to-Peer) არქიტექტურა – განსხვავებული მიდგომა****

📌 **P2P არქიტექტურაში სერვერები თითქმის არ არის** ან საერთოდ არ გამოიყენება.

🔹 მაგივრად, აპლიკაცია მუშაობს **კომპიუტერებს შორის პირდაპირ** — მათ შორის, რომლებიც მომხმარებლებს ეკუთვნით (ლეპტოპები, დესკტოპები და ა.შ.)  
ამ კომპიუტერებს ეძახიან **Peers (თანასწორები)**.

### 🎯 განსხვავება მოკლედ:

|  | **Client-Server** | **P2P (Peer-to-Peer)** |
| --- | --- | --- |
| 📍 მთავარი კვანძი | სერვერი | მომხმარებლების მოწყობილობები |
| 💬 კავშირი | კლიენტი ↔ სერვერი | პირი ↔ პირი |
| 💰 ხარჯი | ძვირი (სერვერები, მონაცემთა ცენტრი) | იაფი (მომხმარებელი იხმარებს თავის მოწყობილობას) |
| 🔒 უსაფრთხოება | უკეთ კონტროლდება | რთულად მართვადი, სუსტი უსაფრთხოება |
| 🔁 შესრულება | სტაბილური, მაგრამ შეიძლება გადატვირთვა | თვითონ იზრდება მომხმარებელთა რაოდენობის ზრდასთან ერთად |
| 📂 მაგალითები | Gmail, Facebook, YouTube | BitTorrent, Skype, Xunlei |

### 🧠 საინტერესო ფაქტი – ჰიბრიდული არქიტექტურა

ზოგი აპლიკაცია იყენებს **ორივე მოდელის შერეულ ვარიანტს**.  
მაგალითად:

* **სერვერი მხოლოდ იმახსოვრებს ვინ ონლაინშია (IP-ები)**
* ხოლო **მესიჯები პირდაპირ გადადის მომხმარებლებს შორის**

მაგალითად: **Skype** ან **ზოგი მესენჯერი**.

პროცესი ქსელში შეტყობინებებს აგზავნის და იღებს შეტყობინებებს პროგრამული ინტერფეისის მეშვეობით, რომელსაც სოკეტი ეწოდება.

* **პროცესი** არის სახლი
* **სოკეტი** არის კარის ჭუჭრუტანა ან კარი
* **ქსელი** არის გზები და ტრანსპორტი, რომლებიც აკავშირებს სახლებს

➡️ როცა ერთი სახლის ბინადარი (პროცესი) უნდა რომ რაღაც გაგზავნოს მეორეში, ის შეტყობინებას გარეთ, კარის ჭუჭრუტანაში (სოკეტში) აწყობს და გზავნის.

➡️ გზები (ქსელი) უძღვება ამ შეტყობინებას მეორე სახლისკენ (მიღების პროცესისკენ), სადაც ის **შეისვლება მეორე სოკეტში**, ანუ მეორე კარში.

➡️ ბოლოს, მეორე სახლის ბინადარი (მიღების პროცესი) კითხულობს შეტყობინებას და რეაგირებს მასზე.

API ანუ სოკეტი

 თითო კომპიუტერზე შეიძლება ბევრი აპლიკაცია მუშაობდეს ერთდროულად.

 **პორტი** ზუსტად გვეუბნება: **“მიპოვნე რომელი პროგრამაა ამ კომპიუტერზე”**

 მაგალითად:

* პორტი **80** = ვებსერვერი (HTTP)
* პორტი **25** = ელფოსტა (SMTP)
* პორტი **443** = დაცული ვებსერვერი (HTTPS)

როდესაც აპლიკაცია აგზავნის მონაცემს სოკეტში, **ტრანსპორტის ფენა** (Transport Layer) პასუხისმგებელია იმაზე, რომ ეს მონაცემი მიაღწიოს მეორე კომპიუტერის სოკეტამდე

## 🚄 ✈️ როგორ ვირჩევ ტრანსპორტის პროტოკოლს?

ინტერნეტში რამდენიმე ტრანსპორტის პროტოკოლი არსებობს (მაგალითად, TCP და UDP). როცა აპლიკაციას ქმნი, უნდა გადაწყვიტო:

👉 **რომელი პროტოკოლი უკეთ შეესაბამება შენი აპლიკაციის მოთხოვნებს?**

📌 ეს ზუსტად ჰგავს იმას, როცა მოგზაურობისთვის ირჩევ:

* **მატარებელი**: უფრო ნელი, მაგრამ ქალაქის ცენტრში გაჩერება, უფრო სტაბილური
* **თვითმფრინავი**: სწრაფი, მაგრამ უფრო რთული დასაწყებად

## 🌐 ტრანსპორტის ფენის 4 ძირითადი სერვისი – მარტივად

### ✅ ****1. სანდო გადაცემა (Reliable Data Transfer)****

📌 ქსელში პაკეტები შეიძლება დაიკარგოს ან დაზიანდეს.

* მაგალითად:
  + პაკეტი გადატვირთულ როუტერში წაიშალოს,
  + ან ბიტები შეიცვალოს გადაცემისას.

🔐 ზოგ აპლიკაციას სჭირდება **სრული სანდოობა** – რომ მონაცემი **სწორად და მთლიანად მივიდეს**.

📍 ამ დროს ვიყენებთ **TCP-ს**, რომელიც:

* გადაამოწმებს პაკეტებს,
* თავიდან აგზავნის დაკარგულ პაკეტებს,
* იცავს მონაცემების მთლიანობას.

🟩 **ვის სჭირდება?**

* ელფოსტა (Gmail)
* ფაილების გადაცემა
* ვებსაიტები
* ბანკები

🔻 თუ ვიყენებთ **UDP-ს**, მონაცემი შეიძლება დაიკარგოს, მაგრამ ზოგ აპლიკაციას ეგ მისაღებია.

### 📶 ****2. გამტარობა (Throughput)****

📌 Throughput = რამდენად სწრაფად შეიძლება მონაცემის გაგზავნა და მიღება

🔹 ზოგი აპლიკაცია საჭიროებს **მინიმუმს throughput-ში** (მაგალითად, 32 kbps მაინც), რომ სწორად იმუშაოს.

🟩 **ვის სჭირდება?**

* ვიდეოზარები, ხმოვანი ზარები
* Zoom, Skype, Teams
* Streaming (YouTube, Netflix)

🟥 თუ არ მიეწოდება საკმარისი გამტარობა:

* ხმა გაწყვეტილი იქნება
* ვიდეო დაიწევს ხარისხში ან გაიყინება

🟡 ზოგი აპლიკაცია კი გამტარობას მარტივად ეგუება — **"elastic" აპლიკაციები**:

* ელფოსტა
* ფაილების გადაცემა
* ვებსაიტების ჩატვირთვა

💬 გამტარობა მათთვის მნიშვნელოვანია, მაგრამ არა გადამწყვეტი.

### ⏱ ****3. დროითი შეზღუდვა (Timing)****

📌 ზოგი აპლიკაცია ითხოვს, რომ პაკეტები **მაქსიმუმ X მილიწამში** მივიდეს.

მაგალითად:  
**100 მილიწამის შიგნით** უნდა მივიდეს ხმოვანი პაკეტი, თორემ შეიგრძნობა პაუზა.

🟩 ვის სჭირდება:

* ინტერნეტ ტელეფონია (VoIP)
* ვიდეოკონფერენცია
* Multiplayer თამაშები
* ვირტუალური რეალობა

🟥 დროის დაგვიანება = რეაქციის დაგვიანება

📍 არარეალურია TCP-სგან ელოდო დროულობას — ამისთვის ხშირად იყენებენ UDP-ს + სპეციალურ ალგორითმებს.

### 🔐 ****4. დაცვა (Security)****

📌 ტრანსპორტის ფენა ზოგჯერ უზრუნველყოფს:

* **დაშიფვრას (Encryption)** — მონაცემები არ იკითხება გზაში
* **მთლიანობის დაცვას (Integrity)** — არავინ ცვლის გზაში
* **იდენტიფიკაციას (Authentication)** — ვიცით, ვინ არის მხარეები

🟩 ვის სჭირდება:

* ბანკები
* ონლაინ გადახდები
* ჩატები (Messenger, WhatsApp)
* დაცული ვებსაიტები (HTTPS)

➡️ მაგალითად, TLS პროტოკოლი მუშაობს TCP-ზე, რომ გააძლიეროს უსაფრთხოება.

## 🧭 შეჯამება ცხრილში:

| **მომსახურება** | **რა უზრუნველყოფს** | **ვის სჭირდება** |
| --- | --- | --- |
| ✅ სანდოობა | TCP | Email, ფაილები, ვებ |
| 📶 გამტარობა | TCP / UDP | ვიდეო/აუდიო ზარები |
| ⏱ დროულობა | UDP + QoS | თამაშები, VoIP, Zoom |
| 🔐 უსაფრთხოება | TLS, SSL | ბანკები, დაცული ვებსაიტები |

**Stateless** (მდგომარეობის გარეშე) ნიშნავს, რომ HTTP პროტოკოლი არ ინახავს ინფორმაციას წინა მოთხოვნების შესახებ, ანუ თითოეული მოთხოვნა (request) სერვერის მიერ დამუშავდება დამოუკიდებლად, წინა კავშირის მდგომარეობის გარეშე.

**1. რა არის HTTP კავშირი?**

HTTP (HyperText Transfer Protocol) არის პროტოკოლი, რომელსაც ვებ-ბრაუზერები და სერვერები იყენებენ ინფორმაციის გადასაცემად, მაგალითად, ვებ-გვერდის მონაცემების გადმოსაცემად. HTTP კავშირი განსაზღვრავს, თუ როგორ ხდება კლიენტის (ბრაუზერის) და სერვერის ურთიერთქმედება. ის შეიძლება იყოს **არამუდმივი** ან **მუდმივი**.

**2. არამუდმივი (Non-Persistent) HTTP კავშირი**

* **რა არის?** არამუდმივი HTTP კავშირის შემთხვევაში, ყოველი მოთხოვნისთვის (request) იქმნება ახალი TCP კავშირი. მაგალითად, თუ ვებ-გვერდი შეიცავს HTML-ს, სურათებსა და CSS ფაილებს, თითოეული ელემენტის ჩასატვირთად ცალკე TCP კავშირი იხსნება.
* **მუშაობის პრინციპი:**
  1. კლიენტი აგზავნის მოთხოვნას (request) სერვერთან.
  2. სერვერი აბრუნებს პასუხს (response).
  3. TCP კავშირი იხურება.
  4. ახალი მოთხოვნისთვის პროცესი მეორდება.
* **უპირატესობები:**
  1. მარტივი განხორციელება.
  2. თითოეული მოთხოვნა დამოუკიდებელია, რაც ამცირებს სერვერის გადატვირთვის რისკს.
* **უარყოფითი მხარეები:**
  1. ყოველი მოთხოვნისთვის ახალი TCP კავშირის შექმნა ზრდის ლატენტურობას (დაყოვნებას).
  2. მეტი რესურსი (CPU, მეხსიერება) იხარჯება.
  3. არაეფექტურია, თუ ვებ-გვერდი მრავალ ფაილს შეიცავს.

**3. მუდმივი (Persistent) HTTP კავშირი**

* **რა არის?** მუდმივი HTTP კავშირი (ჩვეულებრივ HTTP/1.1-ის ნაწილი) საშუალებას აძლევს ერთ TCP კავშირს, გადაიტანოს მრავალი მოთხოვნა-პასუხი, სანამ ის დაიხურება.
* **მუშაობის პრინციპი:**
  1. კლიენტი აგზავნის მოთხოვნას სერვერთან.
  2. სერვერი აბრუნებს პასუხს.
  3. TCP კავშირი ღია რჩება, რათა გამოიყენონ შემდეგი მოთხოვნებისთვის.
  4. კავშირი იხურება მხოლოდ გარკვეული დროის ან მოთხოვნების რაოდენობის შემდეგ.
* **უპირატესობები:**
  1. ამცირებს ლატენტურობას, რადგან არ არის საჭირო TCP კავშირის ხელახლა შექმნა.
  2. ნაკლებ რესურსს მოიხმარს.
  3. უფრო სწრაფია, განსაკუთრებით ვებ-გვერდებისთვის, რომლებიც მრავალ რესურსს მოითხოვს.
* **უარყოფითი მხარეები:**
  1. სერვერს შეიძლება დასჭირდეს მეტი მეხსიერება ღია კავშირების შესანარჩუნებლად.
  2. თუ კავშირი ძალიან დიდხანს რჩება ღიად, შეიძლება გამოიწვიოს სერვერის გადატვირთვა.

**4. ძირითადი განსხვავებები**

| **მახასიათებელი** | **არამუდმივი HTTP** | **მუდმივი HTTP** |
| --- | --- | --- |
| **TCP კავშირი** | ყოველი მოთხოვნისთვის ახალი კავშირი | ერთი კავშირი მრავალი მოთხოვნისთვის |
| **სიჩქარე** | ნელი (მეტი დაყოვნება) | სწრაფი (ნაკლები დაყოვნება) |
| **რესურსების გამოყენება** | მაღალი | დაბალი |
| **გამოყენება** | HTTP/1.0-ისთვის დამახასიათებელი | HTTP/1.1-ისა და უფრო ახალი ვერსიებისთვის |

**5. Pipeling-ის კონცეფცია**

მუდმივი HTTP კავშირის ერთ-ერთი მნიშვნელოვანი თვისებაა **pipelining**, რომელიც საშუალებას აძლევს კლიენტს, გაგზავნოს მრავალი მოთხოვნა ერთდროულად, სანამ სერვერის პასუხს დაელოდება. ეს კიდევ უფრო ამცირებს ლატენტურობას, თუმცა ყველა სერვერი ან ბრაუზერი არ უჭერს მხარს ამ ფუნქციას.

**6. როდის გამოიყენება თითოეული?**

* **არამუდმივი HTTP:** გამოიყენება ძველი სისტემებისთვის ან ისეთ შემთხვევებში, სადაც მოთხოვნები იშვიათია და არ საჭიროებს ხშირ კავშირს.
* **მუდმივი HTTP:** თანამედროვე ვებ-გვერდებისთვის, რომლებიც მოიცავს მრავალ რესურსს (სურათები, CSS, JavaScript), რადგან ეს ზრდის ეფექტურობას.

**7. დასკვნა**

არამუდმივი HTTP კავშირი მარტივი, მაგრამ ნაკლებად ეფექტურია, ხოლო მუდმივი HTTP კავშირი უფრო სწრაფი და რესურსების მხრივ ეფექტურია, რაც მას თანამედროვე ვებ-აპლიკაციებისთვის სასურველს ხდის. HTTP/2 და HTTP/3 კიდევ უფრო აუმჯობესებენ მუდმივი კავშირის ეფექტურობას.

RTT (Round-Trip Time) არის დრო, რომელიც საჭიროა მონაცემთა პაკეტის გასაგზავნად კლიენტიდან სერვერზე და უკან დასაბრუნებლად. ის გამოიყენება ქსელის ეფექტურობისა და ლატენტურობის საზომად. HTTP კავშირების კონტექსტში, RTT განსაკუთრებით მნიშვნელოვანია:

- \*\*არამუდმივი (Non-Persistent) HTTP:\*\* ყოველი მოთხოვნისთვის ახალი TCP კავშირის შექმნა ზრდის RTT-ს, რადგან ხდება TCP-ის სამმხრივი ხელის ჩამორთმევა (three-way handshake) თითოეულ მოთხოვნაზე. ეს ნიშნავს, რომ თითოეული ფაილის (HTML, CSS, სურათი) ჩატვირთვას დამატებითი დრო სჭირდება.

- \*\*მუდმივი (Persistent) HTTP:\*\* RTT-ის გავლენა მცირდება, რადგან ერთი TCP კავშირი გამოიყენება მრავალი მოთხოვნისთვის. ეს ამცირებს ხელის ჩამორთმევისა და კავშირის დამყარების დროს, რაც აჩქარებს ვებ-გვერდის ჩატვირთვას.

\*\*მაგალითი:\*\* თუ RTT არის 100 ms და ვებ-გვერდს აქვს 10 ფაილი:

- არამუდმივი HTTP-ის შემთხვევაში, თითოეული ფაილი მოითხოვს ცალკე კავშირს, ანუ 10 × 100 ms = 1000 ms (გამარტივებული გაანგარიშება).

- მუდმივი HTTP-ის შემთხვევაში, მხოლოდ ერთი კავშირის RTT (100 ms) + მონაცემთა გადაცემის დროა საჭირო, რაც მნიშვნელოვნად ამცირებს ჯამურ ლატენტურობას.

RTT-ის შემცირება მნიშვნელოვანია სწრაფი ვებ-გვერდებისთვის, განსაკუთრებით მობილურ ქსელებში, სადაც ლატენტურობა უფრო მაღალია. HTTP/2 და HTTP/3 კიდევ უფრო ამცირებენ RTT-ის გავლენას multiplexing-ისა და სხვა ოპტიმიზაციების მეშვეობით.

თუ RTT-ის შესახებ კონკრეტული დეტალი ან გაანგარიშება გაინტერესებთ, მითხარეთ!

\*\*სამმხრივი ხელის ჩამორთმევა (Three-Way Handshake)\*\* არის TCP (Transmission Control Protocol) პროტოკოლის მექანიზმი, რომელიც გამოიყენება კლიენტსა და სერვერს შორის სანდო კავშირის დასამყარებლად, სანამ მონაცემთა გადაცემა დაიწყება. ის უზრუნველყოფს, რომ ორივე მხარე მზადაა კომუნიკაციისთვის და სინქრონიზებულია. ეს პროცესი განსაკუთრებით მნიშვნელოვანია HTTP კავშირებში, განსაკუთრებით \*\*არამუდმივი (Non-Persistent)\*\* კავშირების შემთხვევაში, სადაც ყოველი მოთხოვნისთვის ახალი TCP კავშირი იქმნება.

### \*\*სამმხრივი ხელის ჩამორთმევის პროცესი\*\*

სამმხრივი ხელის ჩამორთმევა შედგება სამი ნაბიჯისგან:

1. \*\*SYN (Synchronize):\*\*

- კლიენტი აგზავნის SYN პაკეტს სერვერთან, რომელშიც მითითებულია თავდაპირველი თანმიმდევრობის ნომერი (Sequence Number). ეს არის სიგნალი, რომ კლიენტი სურს კავშირის დამყარება.

- მაგალითად: კლიენტი -> სერვერი: „მინდა კავშირის დამყარება, ჩემი თანმიმდევრობის ნომერია X“.

2. \*\*SYN-ACK (Synchronize-Acknowledgment):\*\*

- სერვერი პასუხობს SYN-ACK პაკეტით, რომელიც ადასტურებს კლიენტის მოთხოვნას (ACK) და აგზავნის საკუთარ თანმიმდევრობის ნომერს (SYN).

- მაგალითად: სერვერი -> კლიენტი: „ვადასტურებ, რომ მივიღე X, ჩემი თანმიმდევრობის ნომერია Y“.

3. \*\*ACK (Acknowledgment):\*\*

- კლიენტი აგზავნის ACK პაკე�。他

System: It looks like the response was cut off. Let me complete the explanation of the \*\*Three-Way Handshake\*\* in Georgian:

### \*\*სამმხრივი ხელის ჩამორთმევის პროცესი (გაგრძელება)\*\*

3. \*\*ACK (Acknowledgment):\*\*

- კლიენტი აგზავნის ACK პაკეტს სერვერთან, რომელიც ადასტურებს სერვერის თანმიმდევრობის ნომრის მიღებას. ამით კავშირი ოფიციალურად დამყარებულია.

- მაგალითად: კლიენტი -> სერვერი: „ვადასტურებ, მივიღე Y, კავშირი მზადაა“.

ამ სამი ნაბიჯის შემდეგ, TCP კავშირი დამყარებულია, და კლიენტი და სერვერი იწყებენ მონაცემთა გაცვლას (მაგ., HTTP მოთხოვნა-პასუხი).

### \*\*რატომ არის სამმხრივი ხელის ჩამორთმევა მნიშვნელოვანი?\*\*

- \*\*სანდოობა:\*\* უზრუნველყოფს, რომ ორივე მხარე (კლიენტი და სერვერი) მზადაა მონაცემთა გადასაცემად.

- \*\*სინქრონიზაცია:\*\* თანმიმდევრობის ნომრები საშუალებას აძლევს მხარეებს, თვალყური ადევნონ მონაცემთა პაკეტების თანმიმდევრობას და თავიდან აიცილონ დაკარგვა ან არეულობა.

- \*\*ეფექტურობა:\*\* ადასტურებს, რომ ქსელი ფუნქციობს და მონაცემთა გადაცემა შესაძლებელია.

### \*\*გავლენა HTTP კავშირებზე\*\*

- \*\*არამუდმივი (Non-Persistent) HTTP:\*\* თითოეული მოთხოვნისთვის (მაგ., HTML, სურათი, CSS) საჭიროა ახალი TCP კავშირი, რაც ნიშნავს, რომ სამმხრივი ხელის ჩამორთმევა ხდება ყოველ ჯერზე. ეს ზრდის \*\*RTT (Round-Trip Time)\*\*-ს და ანელებს ვებ-გვერდის ჩატვირთვას.

- \*\*მუდმივი (Persistent) HTTP:\*\* სამმხრივი ხელის ჩამორთმევა ხდება მხოლოდ ერთხელ, კავშირის დაწყებისას. შემდგომი მოთხოვნები იყენებენ იმავე TCP კავშირს, რაც ამცირებს RTT-ს და აჩქარებს პროცესს.

### \*\*მაგალითი:\*\*

თუ RTT 100 მილიწამია:

- არამუდმივი HTTP-ისთვის, 10 ფაილის ჩასატვირთად შეიძლება დასჭირდეს 10 × (3 × 100 ms) = 3000 ms (თითოეულ ფაილზე სამი RTT, გამარტივებული გაანგარიშებით).

- მუდმივი HTTP-ისთვის, მხოლოდ ერთი სამმხრივი ხელის ჩამორთმევა (3 × 100 ms = 300 ms) + მონაცემთა გადაცემის დროა საჭირო.

### \*\*დასკვნა\*\*

სამმხრივი ხელის ჩამორთმევა აუცილებელია TCP კავშირის სანდოობისთვის, მაგრამ ის ზრდის ლატენტურობას, განსაკუთრებით არამუდმივი HTTP-ის შემთხვევაში. მუდმივი HTTP, HTTP/2 ან HTTP/3 ამცირებენ ამ გავლენას, რადგან ისინი ამცირებენ ხელის ჩამორთმევის საჭიროებას.

თუ გსურთ მეტი დეტალი ან გაანგარიშება, მითხარით!

ძალიან მარტივად, ქართულად:

### \*\*1. Without Parallel Connection (პარალელური კავშირის გარეშე):\*\*

- როცა ვებ-გვერდი იტვირთება, თითოეული ფაილი (HTML, სურათი, CSS) ცალ-ცალკე იგზავნება \*\*ერთი TCP კავშირით\*\*, ერთმანეთის მიყოლებით.

- მაგ.: თუ გვაქვს 3 ფაილი, ჯერ ერთი ჩაიტვირთება, მერე მეორე, მერე მესამე. \*\*ნელია\*\*, რადგან ყველაფერი რიგრიგობით ხდება.

### \*\*2. With Parallel Connection (პარალელური კავშირებით):\*\*

- ერთდროულად \*\*რამდენიმე TCP კავშირი\*\* იხსნება, რათა სხვადასხვა ფაილი (HTML, სურათი, CSS) ერთდროულად ჩაიტვირთოს.

- მაგ.: 3 ფაილი ერთდროულად იტვირთება 3 ცალკე კავშირით. \*\*უფრო სწრაფია\*\*, მაგრამ მეტ რესურსს მოიხმარს.

### \*\*3. Pipelined (ფაიფლაინდი):\*\*

- ერთი TCP კავშირის გამოყენებით, მრავალი მოთხოვნა (request) იგზავნება \*\*ერთდროულად\*\*, სანამ სერვერის პასუხს დაველოდებით.

- მაგ.: 3 ფაილის მოთხოვნა ერთდროულად მიდის ერთ კავშირში, სერვერი ზედიზედ აგზავნის პასუხებს. \*\*სწრაფია\*\* და ნაკლებ რესურსს ხარჯავს.

### \*\*4. Non-Pipelined (არაფაიფლაინდი):\*\*

- ერთ TCP კავშირში მოთხოვნები იგზავნება \*\*რიგრიგობით\*\*. ჯერ ერთი ფაილის მოთხოვნა მიდის, სერვერი პასუხობს, მერე მეორე ფაილის მოთხოვნა, და ა.შ.

- მაგ.: 3 ფაილისთვის 3 ცალკე მოთხოვნა-პასუხის ციკლი ხდება ერთ კავშირში. \*\*ნელია\*\*, რადგან ყოველ ჯერზე ელოდება პასუხს.

### \*\*შეჯამება:\*\*

- \*\*Without Parallel:\*\* ყველაფერი ნელა, ერთი კავშირით, რიგრიგობით.

- \*\*With Parallel:\*\* სწრაფი, მაგრამ მეტი კავშირი = მეტი რესურსი.

- \*\*Pipelined:\*\* სწრაფი, ერთი კავშირით, მოთხოვნები ერთდროულად.

- \*\*Non-Pipelined:\*\* ნელი, ერთი კავშირით, მოთხოვნები რიგრიგობით.

COOKIES

### \*\*რა არის Cookies?\*\*

Cookies (ქუქი-ფაილები) არის პატარა ტექსტური ფაილები, რომლებსაც ვებ-საიტები ინახავენ თქვენს კომპიუტერში, სმარტფონში ან სხვა მოწყობილობაში, როდესაც თქვენ სტუმრობთ მათ. ისინი შეიცავენ ინფორმაციას, რომელიც ეხმარება ვებ-საიტს „დაიმახსოვროს“ თქვენი ქმედებები, პრეფერენციები ან იდენტობა. Cookies ძირითადად გამოიყენება HTTP პროტოკოლის ფარგლებში, რადგან HTTP თავისთავად „მდგომარეობის გარეშეა“ (stateless), ანუ არ ინახავს ინფორმაციას მომხმარებლის წინა ქმედებების შესახებ.

### \*\*Cookies-ის მიზანი\*\*

Cookies-ს რამდენიმე ძირითადი ფუნქცია აქვს:

1. \*\*მომხმარებლის იდენტიფიკაცია:\*\* ინახავს ინფორმაციას, როგორიცაა თქვენი სესიის ID, რათა საიტმა „გაიხსენოს“, რომ თქვენ უკვე შესული ხართ (მაგ., სოციალურ ქსელში).

2. \*\*პრეფერენციების შენახვა:\*\* ინახავს თქვენს არჩევანს (მაგ., ენის პარამეტრი, თემა: ღია/მუქი).

3. \*\*თვალყურის დევნება:\*\* გამოიყენება თქვენი ქცევის ანალიზისთვის, მაგ., რომელი გვერდები ნახეთ ან რა პროდუქტები დაათვალიერეთ.

4. \*\*რეკლამა:\*\* ეხმარება საიტებს, მოგაწოდოთ თქვენს ინტერესებზე მორგებული რეკლამები.

---

### \*\*Cookies-ის ტიპები\*\*

Cookies-ს აქვს სხვადასხვა ტიპი, თითოეულს თავისი დანიშნულება:

1. \*\*სესიის Cookies (Session Cookies):\*\*

- \*\*რა არის?\*\* დროებითი ქუქი-ფაილები, რომლებიც ინახება მხოლოდ მანამ, სანამ ბრაუზერს არ დახურავთ.

- \*\*მაგალითი:\*\* საიტზე შესვლისას თქვენი სესიის შენარჩუნება (რომ არ გადიოდეთ ხელახლა).

- \*\*თავისებურება:\*\* იშლება, როცა ბრაუზერს ხურავთ.

2. \*\*მუდმივი Cookies (Persistent Cookies):\*\*

- \*\*რა არის?\*\* ინახება თქვენს მოწყობილობაში გარკვეული დროით (დღეები, თვეები, წლები).

- \*\*მაგალითი:\*\* „დამიმახსოვრე“ ფუნქცია, როცა საიტზე ავტომატურად შედიხართ.

- \*\*თავისებურება:\*\* აქვს ვადის გასვლის თარიღი.

3. \*\*უსაფრთხო Cookies (Secure Cookies):\*\*

- \*\*რა არის?\*\* მხოლოდ HTTPS კავშირზე მუშაობს, რაც უზრუნველყოფს მონაცემთა დაცვას.

- \*\*მაგალითი:\*\* გამოიყენება ბანკის საიტებზე, სადაც მონაცემთა უსაფრთხოება მნიშვნელოვანია.

4. \*\*HttpOnly Cookies:\*\*

- \*\*რა არის?\*\* ქუქი-ფაილი, რომელსაც მხოლოდ სერვერი ხედავს; JavaScript-ით მიუწვდომელია.

- \*\*მაგალითი:\*\* იცავს საიტს XSS (Cross-Site Scripting) თავდასხმებისგან.

5. \*\*მესამე მხარის Cookies (Third-Party Cookies):\*\*

- \*\*რა არის?\*\* სხვა დომენის მიერ შექმნილი ქუქი-ფაილები, ხშირად რეკლამისთვის.

- \*\*მაგალითი:\*\* Google Ads თვალყურს ადევნებს თქვენს ქცევას სხვადასხვა საიტზე.

- \*\*თავისებურება:\*\* ხშირად ბლოკავს ბრაუზერები კონფიდენციალურობის გამო.

6. \*\*Zombie Cookies (ან Supercookies):\*\*

- \*\*რა არის?\*\* „აღდგება“ მაშინაც კი, თუ მომხმარებელი წაშლის მათ, რადგან ინახება სხვა ადგილებში (მაგ., ბრაუზერის ქეშში).

- \*\*მაგალითი:\*\* გამოიყენება თვალყურის დევნებისთვის, მაგრამ ითვლება არაეთიკურად.

---

### \*\*როგორ მუშაობს Cookies?\*\*

1. \*\*ქუქის შექმნა:\*\*

- როცა საიტს სტუმრობთ, სერვერი აგზავნის „Set-Cookie“ HTTP სათაურს, რომელიც ინახავს ქუქის თქვენს მოწყობილობაში.

- მაგ.: `Set-Cookie: user\_id=12345; Expires=Wed, 22 Jul 2026 11:27:00 GMT; Path=/`

2. \*\*ქუქის გაგზავნა:\*\*

- ყოველ ჯერზე, როცა იმავე საიტს ეწვევით, თქვენი ბრაუზერი აგზავნის ქუქის სერვერზე „Cookie“ სათაურის მეშვეობით.

- მაგ.: `Cookie: user\_id=12345`

3. \*\*სერვერის გამოყენება:\*\*

- სერვერი იყენებს ქუქის, რათა „გაიხსენოს“ თქვენი იდენტობა, პარამეტრები ან ქცევა.

---

### \*\*Cookies-ის სარგებელი\*\*

- \*\*მომხმარებლის გამოცდილება:\*\* აადვილებს საიტის გამოყენებას (მაგ., ავტომატური შესვლა, ენის არჩევანის შენარჩუნება).

- \*\*პერსონალიზაცია:\*\* საიტები გთავაზობენ თქვენს ინტერესებზე მორგებულ შინაარსს.

- \*\*ანალიტიკა:\*\* ეხმარება საიტის მფლობელებს, გაიგონ, როგორ იყენებენ მომხმარებლები მათ საიტს.

- \*\*რეკლამა:\*\* მესამე მხარის ქუქიები საშუალებას აძლევს რეკლამის მიზნობრივად განთავსებას.

---

### \*\*Cookies-ის უარყოფითი მხარეები\*\*

- \*\*კონფიდენციალურობა:\*\* მესამე მხარის ქუქიები თვალყურს ადევნებენ თქვენს ქცევას სხვადასხვა საიტზე, რაც შეიძლება იყოს ინვაზიური.

- \*\*უსაფრთხოება:\*\* თუ საიტი არ არის დაცული (არ იყენებს HTTPS), ქუქიები შეიძლება მოიპარონ.

- \*\*მონაცემთა ბოროტად გამოყენება:\*\* ზოგიერთი საიტი ან რეკლამის განმთავსებელი შეიძლება გამოიყენოს თქვენი მონაცემები არაეთიკურად.

- \*\*შენელება:\*\* დიდი რაოდენობის ქუქიებმა შეიძლება შეანელოს ბრაუზერი.

---

### \*\*Cookies-ის მართვა\*\*

- \*\*ბრაუზერის პარამეტრები:\*\*

- თანამედროვე ბრაუზერები (Chrome, Firefox, Safari) საშუალებას გაძლევთ:

- წაშალოთ ქუქიები.

- დაბლოკოთ მესამე მხარის ქუქიები.

- ჩართოთ „ინკოგნიტო“ რეჟიმი, სადაც სესიის ქუქიები იშლება დახურვისას.

- მაგ.: Chrome-ში: Settings → Privacy and Security → Cookies and other site data.

- \*\*ქუქის თანხმობა:\*\*

- ბევრი საიტი (განსაკუთრებით ევროკავშირში GDPR-ის გამო) ითხოვს თქვენს თანხმობას ქუქიების გამოყენებამდე.

- შეგიძლიათ აირჩიოთ, რომელი ქუქი მიიღოთ (მაგ., მხოლოდ აუცილებელი).

- \*\*ქუქის წაშლა:\*\*

- შეგიძლიათ წაშალოთ ქუქიები ხელით ან გამოიყენოთ ინსტრუმენტები, როგორიცაა CCleaner.

---

### \*\*Cookies და HTTP კავშირები\*\*

- \*\*არამუდმივი (Non-Persistent) HTTP:\*\* ყოველი მოთხოვნისას ქუქი ხელახლა იგზავნება ახალი TCP კავშირის მეშვეობით. ეს ზრდის RTT-ს (Round-Trip Time) და ანელებს პროცესს.

- \*\*მუდმივი (Persistent) HTTP:\*\* ერთი TCP კავშირის გამოყენებით, ქუქიები უფრო სწრაფად გადაიცემა, რადგან არ არის საჭირო ხელახლა ხელის ჩამორთმევა (three-way handshake).

- \*\*Pipelining:\*\* თუ საიტი იყენებს HTTP pipelining-ს, ქუქიების მოთხოვნა-პასუხი უფრო სწრაფია, რადგან მრავალი მოთხოვნა ერთდროულად იგზავნება.

---

### \*\*Cookies-ის უსაფრთხოება\*\*

- \*\*HTTPS:\*\* უსაფრთხო ქუქიები მხოლოდ HTTPS კავშირზე გადაიცემა, რაც ამცირებს მონაცემთა მიტაცების რისკს.

- \*\*HttpOnly:\*\* ხელს უშლის JavaScript-ს, წვდომა ჰქონდეს ქუქის, რაც იცავს XSS თავდასხმებისგან.

- \*\*SameSite ატრიბუტი:\*\*

- „Strict“: ქუქი მხოლოდ იმავე საიტის მოთხოვნებზე იგზავნება.

- „Lax“: ნებადართულია გარკვეული გარე მოთხოვნები (მაგ., ბმულებიდან).

- „None“: ქუქი ყველა მოთხოვნასთან ერთად იგზავნება, მაგრამ საჭიროა HTTPS.

---

### \*\*Cookies-ის მომავალი\*\*

- \*\*კონფიდენციალურობის ცვლილებები:\*\* ბრაუზერები, როგორიცაა Chrome და Firefox, თანდათან ზღუდავენ მესამე მხარის ქუქიებს (მაგ., Chrome-ის „Privacy Sandbox“ ინიციატივა).

- \*\*ალტერნატივები:\*\* საიტები იწყებენ ქუქიების ნაცვლად სხვა ტექნოლოგიების გამოყენებას, როგორიცაა LocalStorage, SessionStorage ან Fingerprinting.

- \*\*კანონმდებლობა:\*\* GDPR (ევროკავშირი) და CCPA (კალიფორნია) აიძულებენ საიტებს, გამჭვირვალე იყვნენ ქუქიების გამოყენების შესახებ.

---

### \*\*მაგალითი\*\*

- \*\*სცენარი:\*\* თქვენ შედიხართ ონლაინ მაღაზიაში:

1. საიტი ინახავს სესიის ქუქის, რათა „დაიმახსოვროს“, რომ შესული ხართ.

2. მუდმივი ქუქი ინახავს თქვენს ენის პრეფერენციას (მაგ., ქართული).

3. მესამე მხარის ქუქი (მაგ., Google Analytics) აგროვებს ინფორმაციას, თუ რომელი პროდუქტები ნახეთ.

---

### \*\*რჩევები მომხმარებლებისთვის\*\*

1. \*\*რეგულარულად წაშალეთ ქუქიები:\*\* ეს ამცირებს თვალყურის დევნებას.

2. \*\*ჩართეთ „Do Not Track“:\*\* თუმცა, ზოგიერთი საიტი შეიძლება იგნორირებას უკეთებდეს.

3. \*\*გამოიყენეთ ინკოგნიტო რეჟიმი:\*\* სესიის ქუქიები იშლება სესიის დასრულებისას.

4. \*\*შეამოწმეთ საიტის ქუქის პოლიტიკა:\*\* გაარკვიეთ, რა მონაცემებს აგროვებენ.

---

თუ გსურთ, რომ განვიხილოთ კონკრეტული ასპექტი (მაგ., ქუქის უსაფრთხოება, მართვა ან HTTP კავშირთან კავშირი), მითხარით, და უფრო დაწვრილებით აგიხსნით!

**WEB CACH**

### \*\*1. ვებ-ქეში (Web Cache)\*\*

- \*\*რა არის?\*\* ვებ-ქეში ინახავს ვებ-გვერდის ნაწილებს (მაგ., სურათები, HTML, CSS) თქვენს მოწყობილობაზე ან შუამავალ სერვერზე, რათა საიტი სწრაფად ჩაიტვირთოს მომავალში.

- \*\*როგორ მუშაობს?\*\*

- როცა საიტს ხსნით, ბრაუზერი ინახავს მონაცემებს (ქეშს).

- მეორედ საიტის გახსნისას ბრაუზერი იყენებს ქეშს, ნაცვლად იმისა, რომ ყველაფერი სერვერიდან გადმოწეროს.

- \*\*სარგებელი:\*\*

- \*\*სიჩქარე:\*\* ნაკლები დრო სჭირდება გვერდის ჩატვირთვას.

- \*\*ტრაფიკის შემცირება:\*\* ამცირებს სერვერის დატვირთვას.

- \*\*მაგალითი:\*\* ხშირად ნანახი საიტის ლოგო ან სურათი იტვირთება თქვენი ბრაუზერის ქეშიდან.

- \*\*უარყოფითი მხარე:\*\* თუ ქეში მოძველებულია, შეიძლება ძველი ვერსია ნახოთ.

---

### \*\*2. Conditional HTTP GET\*\*

- \*\*რა არის?\*\* ეს არის HTTP მოთხოვნა, რომელიც ამოწმებს, შეცვლილა თუ არა საიტის მონაცემები სერვერზე, სანამ ისევ გადმოწერს.

- \*\*როგორ მუშაობს?\*\*

- ბრაუზერი აგზავნის „If-Modified-Since“ ან „ETag“ სათაურს, რომელიც სერვერს ეკითხება: „შეცვლილა თუ არა ეს ფაილი?“

- თუ ფაილი არ შეცვლილა, სერვერი აბრუნებს „304 Not Modified“ და ბრაუზერი იყენებს ქეშს.

- თუ შეცვლილა, სერვერი გადმოგზავნის ახალ ფაილს.

- \*\*სარგებელი:\*\*

- ამცირებს მონაცემთა გადაცემას, თუ ფაილი უცვლელია.

- ზოგავს დროსა და ტრაფიკს.

- \*\*მაგალითი:\*\* თუ საიტის CSS ფაილი არ შეცვლილა, ბრაუზერი იყენებს ქეშს, ნაცვლად ხელახლა გადმოწერისა.

---

### \*\*3. HTTP/2\*\*

- \*\*რა არის?\*\* HTTP-ის განახლებული ვერსია, რომელიც აჩქარებს ვებ-გვერდის ჩატვირთვას.

- \*\*ძირითადი თვისებები:\*\*

- \*\*Multiplexing:\*\* საშუალებას აძლევს ერთ TCP კავშირში რამდენიმე მოთხოვნა-პასუხი ერთდროულად განხორციელდეს (განსხვავებით HTTP/1.1-ისგან, სადაც მოთხოვნები რიგრიგობით მუშავდება).

- \*\*Header Compression:\*\* HTTP სათაურები შეკუმშულია, რაც ამცირებს გადაცემის ზომას.

- \*\*Server Push:\*\* სერვერი აგზავნის მონაცემებს (მაგ., CSS, JavaScript) ჯერ კიდევ სანამ ბრაუზერი მოითხოვს.

- \*\*სარგებელი:\*\*

- უფრო სწრაფი ჩატვირთვა.

- ნაკლები ლატენტურობა (RTT).

- \*\*მაგალითი:\*\* YouTube-ის გვერდი ჩაიტვირთება სწრაფად, რადგან HTTP/2 ერთდროულად გადმოსცემს ვიდეოს, სურათებსა და სკრიპტებს.

---

### \*\*4. HTTP/3\*\*

- \*\*რა არის?\*\* HTTP-ის უახლესი ვერსია, რომელიც იყენებს QUIC პროტოკოლს (UDP-ზე დაფუძნებული) TCP-ის ნაცვლად.

- \*\*ძირითადი თვისებები:\*\*

- \*\*QUIC:\*\* UDP-ზე დაფუძნებული, ამცირებს სამმხრივი ხელის ჩამორთმევის (three-way handshake) დროს.

- \*\*Multiplexing:\*\* ისევე, როგორც HTTP/2, მაგრამ უფრო ეფექტური, რადგან TCP-ის „head-of-line blocking“ პრობლემა აღმოფხვრილია.

- \*\*ჩაშენებული დაშიფვრა:\*\* QUIC-ს აქვს TLS 1.3, რაც ნიშნავს, რომ ყველა კავშირი დაშიფრულია.

- \*\*სარგებელი:\*\*

- კიდევ უფრო სწრაფია, განსაკუთრებით სუსტ ან არასტაბილურ ქსელებში.

- უკეთ მუშაობს მობილურ ინტერნეტზე.

- \*\*მაგალითი:\*\* Google-ის სერვისები (მაგ., Gmail, Google Drive) იყენებენ HTTP/3-ს სწრაფი ჩატვირთვისთვის.

---

### \*\*შედარება მარტივად\*\*

| \*\*მახასიათებელი\*\* | \*\*ვებ-ქეში\*\* | \*\*Conditional HTTP GET\*\* | \*\*HTTP/2\*\* | \*\*HTTP/3\*\* |

|---------------------------|---------------------------|-------------------------|----------------------------|----------------------------|

| \*\*ფუნქცია\*\* | ინახავს მონაცემებს | ამოწმებს ცვლილებებს | აჩქარებს მრავალი ფაილის ჩატვირთვას | კიდევ უფრო სწრაფი, UDP-ზე |

| \*\*სიჩქარე\*\* | აჩქარებს ხელახლა ჩატვირთვას | ამცირებს გადმოწერას | სწრაფი (multiplexing) | ყველაზე სწრაფი |

| \*\*ტექნოლოგია\*\* | ბრაუზერი/სერვერი | HTTP სათაურები | TCP + multiplexing | QUIC (UDP) + TLS |

| \*\*მაგალითი\*\* | სურათის ქეშირება | „304 Not Modified“ | YouTube-ის ჩატვირთვა | Google Drive-ის სიჩქარე |

---

### \*\*შეჯამება\*\*

- \*\*ვებ-ქეში:\*\* ინახავს ფაილებს, რომ საიტი სწრაფად ჩაიტვირთოს.

- \*\*Conditional HTTP GET:\*\* ამოწმებს, საჭიროა თუ არა ფაილის ხელახლა გადმოწერა.

- \*\*HTTP/2:\*\* აჩქარებს ჩატვირთვას multiplexing-ითა და სათაურის შეკუმშვით.

- \*\*HTTP/3:\*\* ყველაზე სწრაფი, QUIC-ის გამოყენებით, იდეალურია თანამედროვე ქსელებისთვის.

თუ გსურთ, რომ რომელიმე თემა უფრო დაწვრილებით განვიხილოთ, მითხარით!

**რა არის SMTP?**

SMTP (Simple Mail Transfer Protocol) არის „ენა“, რომელსაც ინტერნეტი იყენებს, რათა ელ.ფოსტა ერთი ადგილიდან მეორეზე გააგზავნოს. წარმოიდგინე, რომ ეს არის საფოსტო მანქანა, რომელიც წერილებს (ელ.ფოსტებს) აგზავნის სხვა ქალაქებში.

**როგორ მუშაობს SMTP?**

1. **წერილის დაწერა:**
   * თქვენ იყენებთ ელ.ფოსტის პროგრამას (მაგ., Gmail ან Outlook), დაწერეთ წერილი და დააჭირეთ „გაგზავნას“.
2. **SMTP-ის ჩართვა:**
   * თქვენი კომპიუტერი გადასცემს წერილს SMTP სერვერს (მაგ., Gmail-ის სერვერი). ეს სერვერი არის „საფოსტო ოფისი“, რომელმაც იცის, როგორ უნდა გააგზავნოს.
3. **გაგზავნა:**
   * SMTP სერვერი აგზავნის წერილს მიმღების სერვერზე (მაგ., თუ მეგობარს Gmail-ზე გაუგზავნით, მის Gmail სერვერს).
4. **მიღება:**
   * მიმღების სერვერი ინახავს წერილს, სანამ მეგობარი არ გახსნის მას (ამას POP3 ან IMAP ეხმარება).

**მაგალითი მარტივად**

* წარმოიდგინე, რომ მეგობარს Gmail-ზე წერილი გაუგზავნი:
  + შენ წერილს „საფოსტო ყუთში“ (შენი კომპიუტერი) დაწერ, და SMTP-ის მანქანა (Gmail-ის სერვერი) იღებს მას.
  + მანქანა მიჰქრის მეგობრის „საფოსტო ოფისს“ (მის Gmail სერვერს).
  + მეგობარი ხსნის თავის „ჭ mailbox“-ს (ბრაუზერში ან აპში) და ხედავს წერილს.

**ძირითადი თვისებები**

* **მარტივი:** ძალიან მარტივი „ენა“ა, რომელიც მხოლოდ გაგზავნას აკეთებს.
* **უსაფრთხოება:** ჩვეულებრივ არ იცავს მონაცემებს, მაგრამ დღეს TLS/SSL-ით (HTTPS-ის მსგავსად) იცავს.
* **მხოლოდ გაგზავნა:** SMTP მხოლოდ გაგზავნის, მიღებას IMAP ან POP3 აკეთებს.

**რატომაა მნიშვნელოვანი?**

* გარეშე SMTP-ის, შენ ვერ გაგზავნი ელ.ფოსტებს. ის არის ის, რაც აკეთებს, რომ შენი წერილი მეგობარს მიაღწიოს, ნებისმიერი ქვეყნიდან!

**შეცდომები, რომლებიც შეიძლება გაიგო**

* **„SMTP სერვერი უარყო“:** ზოგჯერ SMTP-ის „საფოსტო ოფისი“ უარს ამბობს, თუ პაროლი არასწორია ან ინტერნეტი ცუდია.
* **ნელი გაგზავნა:** თუ ქსელი გადატვირთულია, SMTP-ს შეიძლება დრო დასჭირდეს.

**მაგალითი ცხოვრებიდან**

* ცხოვრებაში, თუ წერილს საფოსტო მანქანით გაგზავნი, SMTP არის ის, ვინც ამ მანქანას მართავს, რომ წერილი სწორ მისამართზე მივიდეს.

### \*\*1. SMTP Handshaking (გ gala)\*\*

- \*\*რა არის?\*\* ეს არის პირველი ნაბიჯი, სადაც შენი კომპიუტერი (ელ.ფოსტის პროგრამა) და SMTP სერვერი (საფოსტო ოფისი) „გიცნობენ ერთმანეთს“.

- \*\*როგორ მუშაობს?\*\* შენი კომპიუტერი ამბობს: „გამარჯობა, მე ვარ [შენი ელ.ფოსტა], და გსურს ჩემი წერილი მიიღო?“. სერვერი პასუხობს: „დიახ, მაგრამ გთხოვ, მიჩვენე შენი პაროლი ან იდენტიფიკაცია“.

- \*\*მაგალითი:\*\* როცა Gmail-ში „გაგზავნა“ დააჭერ, შენი პროგრამა „გაიცნობს“ Gmail-ის სერვერს.

---

### \*\*2. SMTP Transfer of Messages (წერილის გაგზავნა)\*\*

- \*\*რა არის?\*\* ეს არის მეორე ნაბიჯი, სადაც სერვერი იღებს შენს ელ.ფოსტას და იწყებს მის გაგზავნას მიმღებისკენ.

- \*\*როგორ მუშაობს?\*\* სერვერი იღებს შენს წერილს, ადებს მისამართს (მაგ., mze@gmail.com) და „აგზავნის მანქანას“ (სხვა სერვერზე), სადაც მეგობარი იღებს მას.

- \*\*მაგალითი:\*\* შენ გაგზავნი წერილს, და SMTP-ის „მანქანა“ მიჰქრის მეგობრის „საფოსტო ყუთს“.

### \*\*3. SMTP Closure (დახურვა)\*\*

- \*\*რა არის?\*\* ეს ბოლო ნაბიჯია, სადაც კავშირი ჩაიცხება, როცა ყველაფერი დასრულდა.

- \*\*როგორ მუშაობს?\*\* სერვერი ამბობს: „წერილი გავგზავნე, ახლა დავხურავ კავშირს“. შენი კომპიუტერი ამბობს: „კარგი, მადლობა!“.

- \*\*მაგალითი:\*\* როცა Gmail გამოგიჩენს „წერილი გაიგზავნა“, ეს ნიშნავს, რომ SMTP-ის „მანქანამ“ კარგად მუშაობა და კავშირი დახურა.

---

### \*\*მარტივი მაგალითი\*\*

- წარმოიდგინე, რომ მეგობარს წერილი გაუგზავნი:

1. \*\*Handshaking:\*\* შენ და მეგობრის „საფოსტო ოფისი“ ერთმანეთს ამბობთ „გამარჯობა“ და შემოწმებთ, ნამდვილად თქვენ ხართ.

2. \*\*Transfer:\*\* „საფოსტო ოფისი“ იღებს შენს წერილს და მიჰქრის მეგობრის სახლში.

3. \*\*Closure:\*\* როცა წერილი მიაღწია, „საფოსტო ოფისი“ ამბობს: „დასრულდა, ვჩერდები!“.

**DNS**

### \*\*1. DNS-ის საწყისი: რა არის ეს?\*\*

- \*\*ძალიან მარტივად:\*\* DNS არის „ინტერნეტის მრჩეველი“ ან „ტელეფონის წიგნი“. ის გვეხმარება, როცა ვებ-გვერდის სახელს (მაგ., www.google.com) ვეძებთ, გვიჩვენებს, სად არის ეს გვერდი (IP მისამართი, მაგ., 142.250.190.78). IP მისამართი არის „სახლის ნომერი“, რომელსაც კომპიუტერი ამცნობს.

- \*\*მაგალითი:\*\*

- წარმოიდგინე, რომ მეგობარს უნდა გადავაგზავნოთ წერილი, მაგრამ ჩვენ მხოლოდ მისი სახელი „გიორგი“ ვიცით. DNS ცოტა საიდუმლოებით ამბობს: „გიორგის სახლია ქუჩა 12, სახლი 5!“ (IP მისამართი). ამის გარეშე ჩვენ ვერ ვიპოვით, სად არის ის.

### \*\*2. DNS-ის სამსახურები: რას აკეთებს?\*\*

DNS-ს აქვს რამდენიმე „სამუშაო“, რომელიც ყოველდღიურად გვეხმარება. მოდი, ყოველს განვიხილოთ მაგალითებით:

1. \*\*სახელის IP-ად გარდაქმნა (hostname-to-IP-address translation):\*\*

- \*\*რა არის?\*\* ამბობს, თუ რა IP მისამართია ვებ-გვერდის სახელს.

- \*\*მაგალითი:\*\* როცა www.facebook.com-ს ხსნი, DNS ამბობს: „ეს არის 157.240.241.35“. შენი კომპიუტერი ამ მისამართზე მიდის, როგორც მეგობრის სახლში.

2. \*\*სახელის გადარქმევა (host aliasing):\*\*

- \*\*რა არის?\*\* იძლევს სხვა სახელებს ერთი საიტისთვის.

- \*\*მაგალითი:\*\* www.shop.com-ს შეიძლება ჰქონდეს მეორე სახელი, როგორც „store.shop.com“. DNS ამბობს, რომ ორივე ერთია.

- \*\*canonical, alias names:\*\* ერთი ძირითადი სახელი (canonical, მაგ., www.shop.com) და მეორე სახელები (alias, მაგ., store.shop.com).

3. \*\*მეილის დახმარება (mail server aliasing):\*\*

- \*\*რა არის?\*\* ეხმარება ელ.ფოსტებს, რომ იცნონ ერთმანეთი სხვა სახელებით.

- \*\*მაგალითი:\*\* თუ მეილს mail.google.com-ს გაუგზავნი, DNS იცის, რომ ეს იგივეა, რაც gmail.com.

4. \*\*ტვირთის გაყოფა (load distribution):\*\*

- \*\*რა არის?\*\* ნიშნავს, რომ ბევრი ადამიანი ერთდროულად გამოიყენებს საიტს, და DNS ხალხს განაწილებს.

- \*\*მაგალითი:\*\* თუ 1000 ადამიანი youtube.com-ს ხსნის, DNS ამბობს: „100 ადამიანი ამ სერვერზე, 100 სხვაზე“, რათა ყველა სწრაფად ჩაიტვირთოს.

5. \*\*ბევრი მისამართი ერთი სახელისთვის (replicated Web servers):\*\*

- \*\*რა არის?\*\* ერთი სახელი (მაგ., google.com) შეიძლება ჰქონდეს რამდენიმე IP.

- \*\*მაგალითი:\*\* google.com-ს შეიძლება ჰქონდეს 3 IP (1.1.1.1, 2.2.2.2, 3.3.3.3), და DNS არჩევს უახლოესს.

### ❓ კითხვა: ****რატომ არ არის DNS ცენტრალიზებული?****

ანუ:  
რატომ არ ინახება მთელი ინტერნეტის DNS მონაცემები ერთ დიდ ბაზაში, ერთ ადგილას?

### ✅ პასუხი: ****იმიტომ, რომ ეს არ "მასშტაბდება" – ანუ ვერ გაუძლებს დატვირთვას და პრობლემებს გამოიწვევს.****

### მოდი, განვიხილოთ მიზეზები ერთი-ერთზე:

#### 1. 🔴 **Single Point of Failure** – ერთ წერტილში თუ შეიქმნა პრობლემა, მთელ სისტემას დაეტყობა

* თუ DNS სერვერი ერთია და ის გაფუჭდება ან ჰაკერებმა დაბლოკეს – მთელი ინტერნეტი “ჩავარდება”.
* 🔸**მაგალითი:** წარმოიდგინე, რომ ყველას ტელეფონის ნომრები ერთი კარადაში ინახება. თუ ის კარადა დაიკეტა – ვერავის დაურეკავ.

#### 2. 🔵 **Traffic Volume** – უზარმაზარი დატვირთვა

* ყოველდღე მილიარდობით მოთხოვნა იგზავნება DNS სერვერებზე.
  + Comcast → 600 **მილიარდი** მოთხოვნა დღეში
  + Akamai → 2.2 **ტრილიონი** მოთხოვნა დღეში!
* ერთი ცენტრალიზებული სერვერი ამას ვერ გაწვდება.
* 🔸**მაგალითი:** წარმოიდგინე, რომ მთელი ქალაქი ცდილობს ერთ დახლთან პურის ყიდვას. რიგი არ დასრულდება – ყველას დაგვაგვიანდება.

#### 3. 🌍 **Distant Centralized Database** – ერთ ადგილას რომ იყოს, შორს იქნებოდა და ნელა იმუშავებდა

* მომხმარებლები მსოფლიოს ყველა კუთხიდან არიან – თუ DNS მონაცემები მხოლოდ ერთ ქალაქშია (მაგ. ნიუ-იორკში), ტოკიოდან ან თბილისიდან მყოფი მომხმარებლებისთვის ინფორმაცია ნელა მოვა.
* 🔸**მაგალითი:** წარმოიდგინე, რომ ყველა კითხვა უნდა დაუსვა მხოლოდ ერთ მასწავლებელს, რომელიც აფრიკაშია. სანამ პასუხს მიიღებ, დრო გაჭიანურდება.

#### 4. 🛠 **Maintenance** – სისტემის მოვლა და განახლება რთულია

* ერთი დიდი ბაზის მართვა ძვირი და სარისკოა.
* გლობალურად განაწილებული პატარა სერვერები უფრო მარტივად იწესრიგება.
* 🔸**მაგალითი:** ერთი გიგანტური მანქანის შეკეთება რთულია, მაგრამ ბევრი პატარა ველოსიპედის მოვლა – ბევრად მარტივი.

### 🔚 დასკვნა: ****DNS ვერ იქნება ცენტრალიზებული, რადგან****:

* ვერ გაუძლებს ამხელა დატვირთვას,
* თუ მწყობრიდან გამოვა – ყველაფერი გაჩერდება,
* ბევრად ნელა იმუშავებს,
* ძნელად შესანახია.

👉 ამიტომ, DNS სისტემა **გლობალურად არის გადანაწილებული** – ბევრ ადგილას, ბევრ სერვერზეა დაცული. ეს ქმნის სწრაფ, სტაბილურ და დაცულ სისტემას.

## 🚦 როგორ ხდება, როცა ვწერთ www.amazon.com?

შენი კომპიუტერი (DNS client) უნდა გაიგოს, რა **IP მისამართზე** არის ეს საიტი.

მაგრამ მას IP არ ახსოვს და ეკითხება DNS სერვერებს, თანმიმდევრობით ასე:

### 1️⃣ ****კითხვა Root DNS სერვერს****

→ "გთხოვ, მითხარი ვინ იცის .com საიტების მისამართები?"

* Root DNS სერვერებმა იციან, სად არის .com, .org, .edu და სხვების სერვერები.
* ისინი პასუხობენ: "აი, მიდი .com DNS სერვერთან".

📍**მაგალითი:** თითქოს ქუჩაში კითხავ ვინ იცის "კომპანიის" მისამართი, და გეუბნებიან – „მიდი რეგისტრაციის ოფისში, ისინი იციან“.

### 2️⃣ ****კითხვა .com DNS სერვერს****

→ "გთხოვ, მითხარი amazon.com-ს ვინ მართავს?"

* .com სერვერი პასუხობს: "აი, amazon.com-ის DNS სერვერის მისამართი".

📍**მაგალითი:** რეგისტრაციის ოფისში გეუბნებიან – „amazon-ის ოფისი აი აქ არის“.

### 3️⃣ ****კითხვა amazon.com DNS სერვერს****

→ "მითხარი, რა IP აქვს [www.amazon.com-ს](http://www.amazon.xn--com--eer/)?"

* ახლა უკვე მივდივართ იმ ორგანიზაციის სერვერთან, ვისაც მართლა ეს საიტი ეკუთვნის.
* ის პასუხობს: "[www.amazon.com](http://www.amazon.com/) არის მაგალითად → 192.0.2.1".

📍**მაგალითი:** ბოლოს amazon-ის ოფისში გეუბნებიან – „ჩვენი ვებსაიტი აქაა, აი მისი მისამართი“.

## 🧠 ძირითადი დონეები DNS-ში:

| **დონე** | **მაგალითი** | **რას ნიშნავს** |
| --- | --- | --- |
| Root DNS | . (root) | იცის .com, .org, .edu |
| TLD DNS | .com, .org, .edu | იცის კონკრეტული დომენების, მაგ. amazon.com |
| Authoritative DNS | amazon.com, nyu.edu | იცის კონკრეტული ვებსაიტის IP |

## 🔄 შეჯამება — როცა წერ [www.amazon.com](http://www.amazon.com/):

1. შენი კომპიუტერი ეკითხება **root DNS**-ს – "ვინ იცის .com?"
2. მიდის **.com DNS**-თან – "ვინ იცის amazon.com?"
3. მიდის **amazon.com DNS**-თან – "მითხარი [www.amazon.com-ის](http://www.amazon.xn--com--3cruc/) IP მისამართი".
4. იღებს პასუხს და აკავშირებს IP-სთან.

✅ ამიტომ DNS არის სწრაფი, მოქნილი და მთელ მსოფლიოში გადანაწილებული სისტემა.

### 📌 DNS: Root Name Servers

* ✅ **Root name servers** – ესენი არიან ოფიციალური DNS სერვერები, რომლებსაც მიმართავენ მაშინ, როცა სხვა სერვერებმა ვერ უპასუხეს მოთხოვნას.  
  👉 მათ ეძახიან **contact-of-last-resort** – „უკანასკნელი დასაკონტაქტებელი სერვერები“.
* 🌍 ისინი ასრულებენ **სუპერ მნიშვნელოვან როლს ინტერნეტში**.  
  👉 მათი გარეშე ინტერნეტი ვერ იმუშავებს!
* 🔒 **DNSSEC** (Domain Name System Security Extensions) – ეს არის უსაფრთხოების გაფართოება, რომელიც:
  + ამოწმებს, რომ პასუხი ნამდვილად სანდო წყაროდან მოდის (**authentication**),
  + იცავს პასუხს გზაში ცვლილებისგან (**message integrity**)
* 🏛 **ICANN** (Internet Corporation for Assigned Names and Numbers) –  
  ეს არის ორგანიზაცია, რომელიც მართავს Root DNS დომენურ ზონას.  
  👉 ანუ, ისინი პასუხისმგებელნი არიან Root სერვერებზე და DNS-ის გლობალურ წესრიგზე.

## 🌐 **Local DNS Name Servers** – ადგილობრივი DNS სერვერები

### 💡 როგორ მუშაობს?

* როცა კომპიუტერი (ჰოსტი) აგზავნის DNS მოთხოვნას, ის იგზავნება მის **ლოკალურ DNS სერვერზე**
* ეს ლოკალური სერვერი ან:
  + ✅ პასუხობს თვითონ (თუ ქეში აქვს),
  + ❌ ან აგზავნის მოთხოვნას DNS-ის გლობალურ იერარქიაში (Root → TLD → Authoritative)

### 📦 სად ინახება ინფორმაცია?

* ლოკალურ DNS სერვერს აქვს **ქეში** — ბოლოს გამოყენებული სახელებისა და მისამართების ჩამონათვალი
* ქეში შეიძლება იყოს **ძველი/მოძველებული**, ამიტომ ყოველთვის არაა 100% სანდო

### 🧭 ვინ გვაძლევს ლოკალურ DNS სერვერს?

* ყოველი ინტერნეტ მომწოდებელი (ISP) გთავაზობს თავის **DNS სერვერს**
  + მაგალითად: Silknet, Magti, Comcast და სხვ.

### 🖥 როგორ გავიგოთ ჩვენი ლოკალური DNS სერვერი?

* **MacOS:**  
  % scutil --dns
* **Windows:**  
  > ipconfig /all  
  → მოძებნე **"DNS Servers"** ჩანაწერი

**🔵 Recursive Query – "ყველაფერი შენ მოაგვარე!"**

* კლიენტი (მაგ. შენი კომპიუტერი) ეუბნება **ლოკალურ DNS სერვერს**:  
  **"მინდა IP მისამართი www.amazon.com–ის. შენ მოძებნე და დამიბრუნე საბოლოო პასუხი!"**
* ლოკალური DNS სერვერი იწყებს მთელი DNS იერარქიის გადაკითხვას:  
  Root → TLD (.com) → amazon.com → იღებს საბოლოო პასუხს
* **ლოკალური DNS სერვერი პასუხობს კლიენტს უკვე დასრულებული შედეგით**

📍 **მაგალითი:** შენ ეკითხები მეგობარს – „გთხოვ, იპოვე ამ მაღაზიის მისამართი და მეც მომიტანე პასუხი!“ და ის **თავისი ძალებით** ყველგან ეძებს.

**🟢 Iterative Query – "მითხარი ვის ვკითხო!"**

* კლიენტი ან DNS სერვერი ეკითხება სხვა DNS სერვერს:  
  **"იცი www.amazon.com-ის IP?"**
* თუ პასუხი არ იცის, პასუხობს:  
  **"მე არ ვიცი, მაგრამ აი ვის უნდა ჰკითხო – წადი ამ სერვერთან"**
* კლიენტი **თვითონ აგრძელებს** შემდეგ სერვერთან მოთხოვნას

📍 **მაგალითი:** შენ ეკითხები ვიღაცას – „სად არის ეს მაღაზია?“ და ის გეუბნება – „მე არ ვიცი, მაგრამ ნახე იმ ოფისში!“ შენ მიდიხარ და ისევ ეკითხები სხვას.

**🧠 DNS Caching – ქეშირება DNS-ში**

* ✅ როდესაც **ნებისმიერი DNS სერვერი** ერთხელ "ისწავლის" დომენის მისამართს (მაგ. www.amazon.com → 192.0.2.1),  
  ის **შეინახავს** ამ ინფორმაციას თავის ქეშში (მეხსიერებაში)
* ⏱ შემდეგი მსგავსი მოთხოვნისას, **მოაქვს პასუხი ქეშიდან** – სწრაფად, ზედმეტი ძებნის გარეშე
* 🚀 **Caching აუმჯობესებს რეაგირების სისწრაფეს** – არ ხდება ყოველ ჯერზე Root/TLD სერვერებზე კითხვა

**⏳ TTL (Time To Live)**

* ქეშში შენახულ ჩანაწერს აქვს ვადა – **TTL (Time To Live)**
* როცა TTL გავა – ჩანაწერი **წაიშლება** ქეშიდან
* ამის შემდეგ DNS სერვერი თავიდან მოძებნის IP-ს

**⚠️ რისკები ქეშთან დაკავშირებით**

* ქეშში არსებული ჩანაწერი შეიძლება **მოძველებული** იყოს
* თუ ვებგვერდმა შეცვალა თავისი IP მისამართი, ძველი ქეშირებული მონაცემი **გავრცელებული იქნება ინტერნეტში** მანამ, სანამ ყველა TTL არ ამოიწურება

**📌 დასკვნა:**

* DNS არის **best-effort** სისტემა – ცდილობს მაქსიმალურად სწრაფად და სწორად იმუშაოს
* მაგრამ **ზუსტი განახლება არ ხდება მყისიერად** – ქეშის გამო

**Presentation Layer**

**📌 ძირითადი ფუნქციები:**

* ასრულებს მონაცემების გარდაქმნას ისეთ ფორმატში, რომელიც მარტივად აღსაქმელია აპლიკაციის ფენისთვის
* უზრუნველყოფს მონაცემების დაშიფვრასა და გაშიფვრას გადაცემისა და მიღების პროცესში
* ახორციელებს მონაცემების შეკუმშვას, რათა ისინი უფრო სწრაფად და ეფექტიანად გადაეცეს სესიის ფენას

დიახ! აი, სრულად და გარკვევით გადმოცემული ტექსტი გამართული ქართულით, შესაფერისი პრეზენტაციისთვის:

🔹 **წარდგენა (Presentation) ფენა – ფუნქციები**  
წარდგენა წარმოადგენს OSI მოდელის მე-6 ფენას და ასრულებს მნიშვნელოვან როლს, რათა აპლიკაციის ფენამ მიიღოს სწორად დამუშავებული და მისთვის გასაგები ინფორმაცია. ამ ფენას აქვს რამდენიმე ძირითადი ამოცანა:

1. **მონაცემების წარმოდგენის გარდაქმნა**
   * წარდგენის ფენა გარდაქმნის მონაცემებს ისეთი ფორმატიდან, როგორიცაა ბიტები ან სტრიქონები, იმ ფორმატად, რომელსაც იყენებს აპლიკაცია.
   * მაგალითად, თუ ერთ მხარეს გამოყენებულია ASCII და მეორეზე EBCDIC, სწორედ ეს ფენა გარდაქმნის კოდირებას.
2. **დაშიფვრა და გაშიფვრა (Encryption / Decryption)**
   * როდესაც მონაცემები იგზავნება, წარდგენის ფენა ახდენს მათ დაშიფვრას უსაფრთხოების მიზნით.
   * მიღებისას კი იმავე ფენაზე ხდება გაშიფვრა, რათა მომხმარებელმა მიიღოს მონაცემები წაკითხვისთვის ხელმისაწვდომი ფორმით.
3. **შეკუმშვა და გაშლაც (Compression / Decompression)**
   * მონაცემების გაგზავნამდე, ფენა მათ ამცირებს (შეკუმშავს), რათა ტრანსმისია მოხდეს უფრო სწრაფად და ეფექტურად.
   * მიღებისას მონაცემები კვლავ იშლება თავდაპირველ ზომამდე, რათა ზუსტად იქნას წარდგენილი.
4. **სინტაქსის და სემანტიკის უზრუნველყოფა**
   * ფენა უზრუნველყოფს, რომ ორივე მხარეს გამოყენებული მონაცემების სტრუქტურა და მნიშვნელობა იყოს თავსებადი, ანუ მხარეები ერთმანეთს სწორად გაუგონ.

**Ssl/tls**

## 🔐 **რა არის SSL/TLS?**

**SSL (Secure Sockets Layer)** და მისი გაუმჯობესებული ვერსია **TLS (Transport Layer Security)** — არის **დაცული საკომუნიკაციო პროტოკოლები**, რომლებიც იყენება ინტერნეტზე ინფორმაციის დაშიფრული გადაცემისთვის.

📌 თანამედროვე ვებსაიტებზე ვიყენებთ მხოლოდ **TLS-ს** (მაგ. HTTPS://), SSL პრაქტიკულად აღარ გამოიყენება, მაგრამ ტერმინი „SSL“ ჯერ კიდევ ხშირად გვხვდება.

## ✅ **რისთვის გვჭირდება SSL/TLS?**

* 🔒 დაშიფვრა — მონაცემები ვერ წაიკითხება გზაში (მაგ. პაროლები, ბარათის ნომრები).
* ✅ ავთენტიკაცია — გვარწმუნებს, რომ სწორ სერვერთან ვმუშაობთ.
* 📦 მონაცემების მთლიანობა — გზაში არავის შეუძლია მონაცემების შეცვლა.

ამ სურათში ნაჩვენებია როგორ მუშაობს **TLS (Transport Layer Security)** პროტოკოლი კავშირის დასაყენებლად **კლიენტსა და სერვერს** შორის.

ვიწყებ ნაბიჯ-ნაბიჯ მარტივად ახსნას:

## 🔐 TLS კავშირის დამყარება

**TLS არის პროტოკოლი**, რომელიც იცავს შენს პირადობას და ინფორმაციას როცა იყენებ ინტერნეტს (მაგალითად ბრაუზერში შედიხარ ვებსაიტზე HTTPS-ით).

### ▶ 1. ****ClientHello****

* **კლიენტი (მაგ. შენი ბრაუზერი)** ამბობს: "გამარჯობა სერვერო, მინდა დაგიკავშირდე უსაფრთხოდ".
* ამ შეტყობინებაში ის უგზავნის სერვერს:
  + რომელი **TLS ვერსიით** შეუძლია მუშაობა
  + რომელ **შიფრებს** (ალგორითმებს) იყენებს დაშიფვრისთვის
  + აგზავნის „random“ რიცხვს (შემთხვევით რიცხვს), რომელიც საჭიროა დაშიფვრისთვის.

### ▶ 2. ****ServerHello + სერტიფიკატი****

* **სერვერი პასუხობს** კლიენტს:
  + ირჩევს დაშიფვრის ალგორითმს (ერთ-ერთს იმათგან რაც კლიენტმა შესთავაზა)
  + აგზავნის თავის **სერტიფიკატს** (რომელშიც არის საჯარო გასაღები და მტკიცებულება რომ ნამდვილი სერვერია)
  + აგზავნის „random“ რიცხვს თავის მხარესაც
  + ამბობს: "მზად ვარ უსაფრთხო კავშირის დასაწყებად".

### ▶ 3. ****ClientKeyExchange****

* კლიენტი იღებს სერვერის სერტიფიკატს და საჯარო გასაღებით ქმნის **საიდუმლო გასაღებს** (shared secret).
* ეს საიდუმლო გასაღები მხოლოდ კლიენტმა და სერვერმა იციან და ის გამოიყენება შიფრაციისთვის.
* კლიენტი აგზავნის:
  + ამ საიდუმლო გასაღებს, დაშიფრულად.
  + ChangeCipherSpec: „ახლა უკვე ვიწყებთ დაშიფრულ კომუნიკაციას“
  + Finished: ბოლო შეტყობინება ამ ეტაპზე, უკვე დაშიფრულია.

### ▶ 4. ****ChangeCipherSpec + Finished (სერვერიდან)****

* სერვერი პასუხობს:
  + ChangeCipherSpec: „ჰო, მეც ვიწყებ დაშიფრულად საუბარს“
  + Finished: ბოლოჯერ ამოწმებს, რომ ყველაფერი სწორად მოხდა.

### 🕐 დრო: TLS 110 ms

* ეს ყველაფერი ხდება ძალიან სწრაფად — დაახლოებით **110 მილიწამში**.

### ✅ და ბოლოს?

ახლა კლიენტი და სერვერი მზად არიან **დაშიფრული და უსაფრთხო კავშირის** დასაწყებად. მონაცემები ვეღარ მოუსმენს მესამე პირი (ჰაკერი, მზვერავი და ა.შ.).

## 🔒 რისი დაცვა ხდება TLS-ით?

* **კონფიდენციალურობა** (შინაარსს ვერ წაიკითხავენ)
* **ავთენტურობა** (ვერ გააყალბებენ სერვერს)
* **მთლიანობა** (შეტყობინება არ დააზიანდება ან შეიცვლება გზაში)

**Session Layer**

რა თქმა უნდა! აქ არის **Session Layer-ის (სესიის ფენა) Deep Dive** — მარტივად, გასაგებად და თან ვრცლად:

## 🧩 რას ნიშნავს Session Layer?

**Session Layer** არის OSI მოდელის **მეხუთე ფენა** (Layer 5), რომელიც მართავს "სესიის" ანუ **კომუნიკაციის აქტიურ ურთიერთობას ორ მოწყობილობას შორის**.

ანუ როდესაც ერთი აპლიკაცია მეორეს უკავშირდება (მაგ. Zoom ზარი, ვიდეო სტრიმინგი, ან ფაილის გადაცემა), ეს ფენა უზრუნველყოფს, რომ:

* კავშირი სწორად და წესრიგიანად დაიწყოს
* მთელი კომუნიკაციის განმავლობაში ყველაფერი სინქრონში იმუშაოს
* და როცა ყველაფერი დასრულდება — კავშირი სწორად დაიხუროს

## 📌 მთავარი ფუნქციები მარტივად:

### 1. ****სესიის დამყარება (Session Establishment)****

ეს ფენა აყალიბებს "სესიას" ანუ დროებით კომუნიკაციას ორ მხარეს შორის, როგორიცაა:

* კლიენტი ↔ სერვერი
* მომხმარებელი ↔ აპლიკაცია

მაგალითად: როცა Zoom-ზე უკავშირდები სხვის კამერას, session layer აშენებს ამ დიალოგს.

### 2. ****დიალოგის კონტროლი (Dialog Control)****

ეს ფენა წყვეტს — **ვინ როდის ლაპარაკობს**.

* თუ სისტემები მუშაობენ **half-duplex-ით** (ერთი ლაპარაკობს, მეორე უსმენს) — session layer აქ იყენებს ტოკენებს ("ჩემი ჯერია!")
* თუ **full-duplex-ით** მუშაობენ (ორივე ლაპარაკობს ერთდროულად) — ის ამასაც უჭერს მხარს

### 3. ****სინქრონიზაცია (Synchronization)****

სესიის ფენა მონაცემების ნაკადში რთავს სპეციალურ **სინქრონიზაციის წერტილებს (checkpoints)**. ეს ნიშნავს:

* თუ კავშირი გაწყდება, შენ არ დაკარგავ მთელ კომუნიკაციას
* შეგიძლიათ გააგრძელოთ იქიდან, სადაც გაჩერდა

მაგალითად: ვიდეოს ყურებისას თუ ინტერნეტი გაწყდა, როცა დაუბრუნდები, სისტემამ იცის ზუსტად საიდან განაახლოს.

### 4. ****რესინქრონიზაცია (Resynchronization)****

თუ მოხდა გაუმართაობა — session layer ცდილობს **სესია აღადგინოს იქიდან**, სადაც ყველაფერი სწორად იყო.

### 5. ****აქტივობების მენეჯმენტი (Activity Management)****

დატა იყოფა დამოუკიდებელ ნაწილებად (activities), რაც კავშირს მოქნილს და სწორად სამართავს ხდის.

### 6. ****სესიის დახურვა (Session Termination)****

როცა კომუნიკაცია სრულდება, session layer უზრუნველყოფს **დაცულად დახურვას**, ისე რომ არაფერი დაიკარგოს.

## 🛠️ როგორ მუშაობს Session Layer?

1. კლიენტი აგზავნის მოთხოვნას → სერვერთან სესიის დასაწყებად
2. ხდება ავთენტიკაცია (შესაძლოა პაროლით, სერტიფიკატით და ა.შ.)
3. იდგმება წესები: როგორ ილაპარაკებენ (token-based თუ simultaneous)
4. დაიდება სინქრონიზაციის წერტილები
5. გადაეცემა მონაცემები
6. დაიხურება სესია

## 🔐 პროტოკოლები (Session Layer-ის პროტოკოლები)

ეს ფენა იყენებს სპეციფიკურ პროტოკოლებს, რომ უზრუნველყოფილ იქნეს კომუნიკაცია. რამდენიმე მაგალითი:

| **პროტოკოლი** | **აღწერა** |
| --- | --- |
| **ADSP (AppleTalk Data Stream Protocol)** | Apple-ის ქსელში სესიის დამყარებისთვის |
| **RTCP** | იყენებს აუდიო/ვიდეო სტრიმინგში ხარისხის კონტროლისთვის |
| **PPTP** | VPN-ის სესიის მართვისთვის |
| **PAP** | მომხმარებლის ავთენტიკაციისთვის (პაროლით) |
| **RPC (Remote Procedure Call)** | ერთ აპლიკაციას აძლევს საშუალებას, იმუშაოს სხვა კომპიუტერის ფუნქციებთან |
| **SDP** | უფრო სწრაფი კავშირისთვის RDMA ქსელზე (data center-ებში ხშირად გამოიყენება) |

## 📊 მაგალითები მარტივად:

* Zoom-ზე როცა საუბრობ — Session Layer წყვეტს ვინ როდის ლაპარაკობს
* ვიდეო თამაშში, სადაც ორ მოთამაშეს შორის ხმოვანი კავშირია
* რეზერვაცია ან ბანკის სისტემა, სადაც თითოეული მომხმარებელი მართავს თავის სესიას დამოუკიდებლად

## 🔄 რა განსხვავებაა Transport Layer-სა და Session Layer-ს შორის?

| **Transport Layer** | **Session Layer** |
| --- | --- |
| აწვდის მონაცემებს აპლიკაციამდე | მართავს "სესიის" ლოგიკას |
| TCP, UDP | PAP, RPC, RTCP და სხვა |
| მართავს კონექციის შექმნას და დახურვას | მართავს დიალოგს, სინქრონს და აღდგენას |

## ✅ შეჯამება მარტივად:

* **Session Layer** მართავს აქტიური კავშირების ლოგიკას
* ქმნის, მართავს და დახურავს სესიებს
* უზრუნველყოფს სინქრონიზაციასა და აღდგენას
* განსაკუთრებით გამოსადეგია აპლიკაციებში, სადაც უნდა მიეთითოს "ვინ ლაპარაკობს", "როდის გაწყდა", "საიდან გავაგრძელო"

თუ გინდა ამაზე პრეზენტაციის სლაიდებიც გაგიმზადო ან უფრო მოკლე შინაარსი გინდა სლაიდებისთვის, სიამოვნებით დაგეხმარები — მითხარი 🧠💡

**TRANSPORT LAYER**

მოდით, მარტივად და გასაგებად აგიხსნე **Multiplexing** და **Demultiplexing**, რომლებიც დაკავშირებულია **Transport Layer-სთან** (გადაცემის ფენასთან).

## 🔀 Multiplexing და Demultiplexing – მარტივად

### ▶️ ****Multiplexing**** — გაგზავნა:

***"მრავალი აპლიკაციის მონაცემის ერთ არხზე გადაგზავნა"***

როცა შენს კომპიუტერზე რამდენიმე აპლიკაცია ინტერნეტს იყენებს (მაგალითად, ბრაუზერი, Spotify და Zoom), ყველა მათგანი მონაცემებს აგზავნის ერთნაირად — **IP მისამართისა და Transport Layer პროტოკოლის (TCP/UDP) მეშვეობით.**

მაგრამ როგორ გაიგებს ქსელი, რომელი მონაცემი ეკუთვნის რომელ აპლიკაციას?

აქ ჩნდება Multiplexing:

* Transport layer სხვადასხვა აპლიკაციებისთვის სხვადასხვა **port ნომერს** იყენებს.
* მაგალითად:
  + ბრაუზერი იყენებს 443 პორტს (HTTPS)
  + ელ. ფოსტა იყენებს 25 პორტს (SMTP)
  + Zoom შესაძლოა გამოიყენოს 8801 პორტი და ა.შ.

Transport Layer ყველა აპლიკაციის მონაცემს აერთიანებს ერთ „გზავნად“ და გზავნის IP-ზე TCP/UDP headers-ით.

### ⬅️ ****Demultiplexing**** — მიღება:

***"მიღებული მონაცემების გაყოფა სწორ აპლიკაციებზე"***

როცა სერვერი ან სხვა მოწყობილობა იღებს მონაცემებს, transport layer უნდა მიხვდეს:

* რომელი პაკეტი რომელ აპლიკაციას ეკუთვნის?
* ეს ხდება **port ნომრების გამოყენებით** (source port, destination port).

**Demultiplexing** გაშიფრავს მონაცემებს და სწორად გადააწვდიდა კონკრეტულ აპლიკაციას.

## 🔢 მაგალითი

### შენს კომპიუტერზე:

* YouTube → ბრაუზერი → Port 443
* Spotify → App → Port 4070

**Multiplexing**: ბრაუზერის და Spotify-ის მონაცემები ერთდროულად გადის ქსელში სხვადასხვა პორტებით.

### სერვერზე:

**Demultiplexing**: მიღებული პაკეტები თავიანთი პორტის მიხედვით იგზავნება შესაბამის სერვისზე.

## 📌 დასამახსოვრებლად:

| **მოქმედება** | **რას ნიშნავს** | **სად ხდება** |
| --- | --- | --- |
| **Multiplexing** | მონაცემების ერთდროულად გაგზავნა ერთ ხაზზე | გაგზავნისას |
| **Demultiplexing** | მიღებული მონაცემების სწორად დახარისხება | მიღებისას |

### 🔹 რა არის Checksum?

**Checksum** არის რიცხვი, რომელიც გამოიყენება იმის შესამოწმებლად, გაფუჭდა თუ არა მონაცემები გადაცემისას.

ის არის:

* პატარა რიცხვი (მაგ. 16 ბიტი),
* გამომუშავებული მონაცემებისგან,
* იძლევა საშუალებას მიმღებმა შეამოწმოს, სწორად მივიდა თუ არა მონაცემი.

## 🔹 როგორ მუშაობს Checksum?

### 1. ****გაგზავნის მხარე (Sender):****

* იღებს მთელ მონაცემებს (data),
* იყენებს სპეციალურ ალგორითმს (მაგ. one's complement sum),
* ითვლის Checksum-ს,
* გზავნის მონაცემებთან ერთად.

### 2. ****მიმღები მხარე (Receiver):****

* იღებს მონაცემებს და Checksum-ს,
* თავიდან ითვლის Checksum-ს მიღებული მონაცემებიდან,
* ადარებს მიღებულ Checksum-ს გამოთვლილს:
  + თუ ემთხვევა ✅ → მონაცემები არ დაზიანებულა.
  + თუ არ ემთხვევა ❌ → მონაცემები დაზიანდა გზაში.

## 🕒 Timers — ანუ რა?

**Timer**-ები არის **დროის მექანიზმები**, რომლებიც განსაზღვრავენ, როდის უნდა მოხდეს პაკეტის ხელახლა გაგზავნა, ან სხვა მოქმედება, თუ პასუხი დროულად არ მივიღეთ.

## ✅ რატომ ვიყენებთ Timer-ებს?

1. **დაკარგული პაკეტების აღმოჩენა** — თუ პასუხი არ მოვიდა დროულად, ვხვდებით რომ პაკეტი დაიკარგა.
2. **ხელახლა გაგზავნა** — ველოდებით განსაზღვრულ დროს (timeout-ს) და თუ არ მივიღეთ პასუხი, ვგზავნით ხელახლა.
3. **Congestion control** — Timer-ები ეხმარება ქსელში გადატვირთვის აღმოჩენას და მართვას.

## 📥 როგორ მუშაობს Timers TCP-ში (მაგალითი):

### 1. ****Sender**** აგზავნის სეგმენტს

→ ააქტიურებს **retransmission timer-ს**

### 2. თუ ****ACK**** (დადასტურება) დროულად მოვიდა:

→ timer გაუქმდება — ყველაფერი კარგადაა.

### 3. თუ ****ACK არ მოვიდა დროულად****:

→ დრო ამოიწურება → ვთვლით, რომ პაკეტი დაიკარგა → ხელახლა იგზავნება პაკეტი.

## 📊 ძირითადი Timer-ები TCP-ში

| **Timer-ის სახელი** | **აღწერა** |
| --- | --- |
| ⏱ **Retransmission Timer** | დალოდების დრო ACK-ზე. ყველაზე მთავარი timer |
| ⌛ **Persist Timer** | გამოიყენება თუ ფანჯრის ზომა (window size) არის 0 – პაკეტების გაგზავნა დროებით შეჩერდება |
| ⏳ **Keepalive Timer** | იძახებს "ცოცხალი ხარ?" როცა დიდი ხანი არაა კავშირი |
| 🚫 **Time-Wait Timer** | როცა კავშირი იხურება, კიდევ ცოტა ხანს აკავებს რესურსს დაცულად |

## 🧠 მარტივი მაგალითი:

წარმოიდგინე ასე:

* შენ მეგობარს უგზავნი წერილს 📨
* გაწერია დრო: "თუ 5 წუთში არ მიპასუხებს, თავიდან გავუგზავნი"
* ეს არის Timer 🕒
* თუ გიპასუხა დროზე — ყველაფერი რიგზეა ✅
* თუ არა — ხელახლა აგზავნი ❗

## 💡 დასასრული — რატომ არის ეს მნიშვნელოვანი?

ტრანსპორტ ფენის Timer-ები უზრუნველყობენ:

* საიმედო კომუნიკაციას
* მონაცემთა დაკარგვის აღდგენას
* ქსელის ეფექტურ მართვას

გინდა ეს PowerPoint სლაიდებადაც გაგიჩინო ან გრაფიკულად აგიხსნა? 😊

აქ არის შენი ტექსტის უბრალო თარგმანი ქართულად:

**Transmission Control Protocol (TCP)** — ეს არის კავშირისზე ორიენტირებული კომუნიკაციის პროტოკოლი, რომელიც სხვადასხვა მოწყობილობას შორის ქსელზე შეტყობინებების გადაცვლაში ეხმარება. ის TCP/IP პროტოკოლთა ერთ-ერთი მთავარი ნაწილია. **OSI მოდელში**, TCP მუშაობს **ტრანსპორტის ფენაში (Layer 4)**. ის მდებარეობს აპლიკაციისა და ქსელის ფენებს შორის და უზრუნველყოფს მონაცემების საიმედო მიწოდებას. TCP თანამშრომლობს **IP პროტოკოლთან**, რომელიც განსაზღვრავს, თუ როგორ იგზავნება მონაცემთა პაკეტები კომპიუტერებს შორის.

TCP ამყარებს საიმედო კავშირს გამგზავნსა და მიმღებს შორის **სამეტაპიანი ხელმოწერის** (SYN, SYN-ACK, ACK) მეშვეობით, ხოლო კავშირის სწორად დასახურად იყენებს **ოთხეტაპიან პროცედურას** (FIN, ACK, FIN, ACK).

TCP უზრუნველყოფს:

* შეცდომის გარეშე, თანმიმდევრულად მონაცემების მიწოდებას,
* მიღების დადასტურებას (ACK),
* გადატვირთვის თავიდან აცილებას მიმღების ბუფერის ზომის მიხედვით გადაცემის სიჩქარის დაქვითვით,
* ქსელის გადატვირთვის რეგულირებას ისეთი ალგორითმებით, როგორიცაა Slow Start, Congestion Avoidance, Fast Retransmit და Fast Recovery.

TCP-ის თავსართი იყენებს **ჩეკსამს** (checksum), რათა აღმოაჩინოს დაზიანებული მონაცემები და საჭიროების შემთხვევაში მოითხოვოს მათი ხელახლა გაგზავნა.

TCP გამოიყენება ისეთ აპლიკაციებში, სადაც საჭიროა საიმედო და თანმიმდევრული მონაცემების გადაცემა, როგორიცაა ვებ-ბრაუზინგი, ელფოსტა და დისტანციური წვდომა.

**Internet Protocol (IP)**

Internet Protocol (IP) — ეს არის მეთოდი, რომელიც გამოიყენება მონაცემების ერთი მოწყობილობიდან მეორეზე გადაგზავნისთვის ინტერნეტში. ის წარმოადგენს წესების ნაკრებს, რომელიც განსაზღვრავს, როგორ იგზავნება და მიიღება მონაცემები. IP პასუხისმგებელია მონაცემთა პაკეტების **მისამართზე მიყვანასა და მარშრუტიზაციაზე**. ყოველი მოწყობილობა ფლობს უნიკალურ **IP მისამართს**, რაც მას ეხმარება სხვა მოწყობილობებთან ურთიერთობაში.

**როგორ მუშაობს TCP?**

TCP ამტვრევს მონაცემებს მცირე ნაწილებად (პაკეტებად) და შემდეგ მეორე ბოლოზე აწყობს ისევ მთლიანად, რათა დარწმუნდეს, რომ შეტყობინება სრულად და დაზიანების გარეშე მიაღწია დანიშნულების ადგილს. მონაცემების მცირე ნაწილებად გაგზავნა უფრო ეფექტურია, ვიდრე ერთიანად მთელი ინფორმაციის გაგზავნა.

ამ პატარა პაკეტებს შეუძლიათ იმოძრაონ სხვადასხვა მარშრუტით, თუ რომელიმე გზა გადატვირთულია, თუმცა საბოლოო დანიშნულება მაინც იგივეა.

**მაგალითი TCP-ზე მუშაობის**

როდესაც მომხმარებელი ითხოვს ვებგვერდს ინტერნეტში, სადღაც მსოფლიოში სერვერი ამ მოთხოვნას ამუშავებს და უგზავნის HTML ფაილს. სერვერი იყენებს HTTP პროტოკოლს, რომელიც TCP-ს ეუბნება, რომ დაადგინოს კავშირი და გაგზავნოს HTML ფაილი.

შემდეგ TCP ამტვრევს ფაილს პატარა პაკეტებად და გადასცემს მათ IP-ს, რომელიც მათ აგზავნის მომხმარებლამდე სხვადასხვა მარშრუტით.

TCP მომხმარებლის მოწყობილობაზე ელოდება, სანამ ყველა პაკეტი არ მივა და შემდეგ აგზავნის მიღების დადასტურებას.

**TCP-ის მახასიათებლები**

* **სეგმენტების ნომრებით აღნიშვნა** – TCP აკონტროლებს გაგზავნილი და მიღებული სეგმენტების მიმდევრობას ნომრების მინიჭებით.
* **კავშირისზე ორიენტირებულობა** – კავშირი არსებობს სანამ მონაცემთა გადაცემა არ დასრულდება.
* **ორმხრივი კომუნიკაცია (Full Duplex)** – მონაცემები შეიძლება ორივე მიმართულებით გადაიცეს ერთდროულად.
* **გადაცემის სიჩქარის კონტროლი (Flow Control)** – აგრძელებს გადაცემას იმ სიჩქარით, რასაც მიმღები შეძლებს.
* **შეცდომების კონტროლი (Error Control)** – თუ სეგმენტი დაზიანდა ან დაიკარგა, TCP თავიდან აგზავნის.
* **გადატვირთვის კონტროლი (Congestion Control)** – TCP აკონტროლებს ქსელის გადატვირთვის დონეს.

**TCP-ის უპირატესობები**

* საიმედოა
* უზრუნველყოფს შეცდომების აღმოჩენასა და გამოსწორებას
* აქვს სიჩქარის კონტროლი
* უზრუნველყოფს მონაცემების სწორ მიმდევრობით მიწოდებას
* ფართოდ გამოყენებულია და სტანდარტიზებულია (IETF)
* მუშაობს IP-სთან ერთად

**TCP-ის ნაკლოვანებები**

* TCP განკუთვნილია ფართომასშტაბიანი ქსელებისთვის, რაც პატარა ქსელებში პრობლემურია
* მუშაობს რამდენიმე ფენაზე, რაც ამცირებს სიჩქარეს
* მხოლოდ TCP/IP სტეკთან თავსებადია — Bluetooth-თან, მაგალითად, ვერ იმუშავებს
* შემუშავებული დაახლოებით 30 წლის წინ და მას შემდეგ მნიშვნელოვნად არ შეცვლილა

**🧱 ტრანსპორტის ფენის (Transport Layer – Layer 4) ფუნქციები:**

**✅ მონაცემების გამჭვირვალე გადაცემა ბოლო წერტილებს შორის (End Hosts)**

ტრანსპორტის ფენა უზრუნველყოფს მონაცემების გადაცემას ერთი მოწყობილობიდან მეორეზე ისე, რომ მომხმარებელი დეტალებს ვერ ამჩნევს — ეს გადაცემა "გამჭვირვალეა".

**🛠️ სერვისები, რომლებსაც აპლიკაციებისთვის შეიძლება TCP ან UDP უზრუნველყოფდეს (ან არა):**

* **საიმედო მონაცემთა გადაცემა** – დარწმუნება, რომ ყველაფერი მივიდა სრულად და სწორად
* **შეცდომის აღმოფხვრა (Error Recovery)** – დაზიანებული ან დაკარგული მონაცემების თავიდან გაგზავნა
* **მონაცემთა თანმიმდევრობა (Data Sequencing)** – ინფორმაციის სწორად დალაგება მიღებისას
* **ნაკადების კონტროლი (Flow Control)** – გადაცემის სიჩქარის მორგება მიმღების შესაძლებლობაზე

📌 **TCP უზრუნველყოფს ამ ყველაფერს**, ხოლო **UDP — არა** (მარტივი, სწრაფია, მაგრამ ნაკლებად სანდო).

**🔢 Layer 4 მისამართების მინიჭება (Port Numbers – პორტების ნომრები):**

პორტები აუცილებელია, რომ განსაზღვრო **რომელი აპლიკაცია** მიიღებს მონაცემებს.

მაგალითად:

* ვებსერვერი იყენებს პორტს **80** (HTTP)
* FTP იყენებს პორტს **21**
* და ასე შემდეგ...

**🧭 Layer 4 პორტების გამოყენებით შესაძლებელია:**

1. **აპლიკაციის ფენის პროტოკოლის იდენტიფიცირება** – მაგალითად, პორტი 80 → HTTP
2. **სესიის მულტიპლექსირება** – ერთი კომპიუტერი ერთდროულად ბევრ სესიას ამუშავებს სხვადასხვა პორტით (მაგ: ბევრი საიტის გახსნა ერთდროულად).

**📊 პორტების დიაპაზონები (IANA-ს მიერ განსაზღვრული):**

| **პორტის ტიპი** | **დიაპაზონი** | **აღწერა** |
| --- | --- | --- |
| **Well-known (ცნობილი)** | 0 – 1023 | სისტემის ძირითადი სერვისები (HTTP, FTP, DNS...) |
| **Registered (რეგისტრირებული)** | 1024 – 49151 | პროგრამებისთვის, რომლებიც დარეგისტრირებულია IANA-ში |
| **Ephemeral/Private/Dynamic** | 49152 – 65535 | დროებითი პორტები კლიენტის მხარეს |

**🎯 მაგალითები ქვედა კუთხიდან:**

1. (TCP): Src: 50000 Dst: 80  
   – კლიენტი იყენებს პორტს **50000** და უკავშირდება სერვერს **პორტზე 80 (HTTP)**
2. (TCP): Src: 55000 Dst: 80  
   – კლიენტის სხვა სესია, დროებითი პორტით **55000**, ისევ სერვერზე **პორტზე 80**
3. (TCP): Src: 60000 Dst: 21  
   – კლიენტი იყენებს პორტს **60000** და უკავშირდება FTP სერვერს **პორტზე 21**

გავხსნათ ეს ყველაფერი მარტივად და ქართულად:

**TCP არის "კავშირზე ორიენტირებული" პროტოკოლი**  
→ სანამ ერთ კომპიუტერი მეორესთან მონაცემებს გაუგზავნის, ისინი პირველ რიგში ერთმანეთთან "კავშირს ამყარებენ" – ანუ ელაპარაკებიან ერთმანეთს და ეთანხმებიან კომუნიკაციის დაწყებაზე.  
→ კავშირი რომ ერთხელ დადგინდა, მხოლოდ მაშინ იწყება მონაცემების გაცვლა.

**TCP უზრუნველყოფს საიმედო კომუნიკაციას**  
→ როცა ერთი კომპიუტერი მეორეს მონაცემს უგზავნის, მეორე უნდა დაადასტუროს, რომ მიიღო ("Acknowledgment" – მიღების დასტური).  
→ თუ მიღების დასტური არ მოვიდა, იგივე მონაცემი თავიდან იგზავნება.

**TCP უზრუნველყოფს სწორ თანმიმდევრობას (Sequencing)**  
→ TCP-ში არსებობს რიგის ნომრები, რომლებიც ეხმარება მიმღებს, დაალაგოს ნაწილებად გამოგზავნილი მონაცემები სწორად – თუნდაც თუ ისინი არასწორ თანმიმდევრობით მივიდნენ.

**TCP უზრუნველყოფს ნაკადის კონტროლს (Flow Control)**  
→ მიმღები კომპიუტერი ეუბნება გამგზავნს, რომ დააჩქაროს ან შეანელოს გაგზავნა, რათა არ გადატვირთოს თავისი სისტემა.

შეგიძლია ასე დაიმახსოვრო:  
TCP = დაინტერესებული მეგობარი, რომელიც ჯერ გკითხავს მზად ხარ თუ არა საუბრისთვის (კავშირი), დაგელოდება რომ ყველაფერი გაიგო (საიმედოობა), გეტყვის რა და როდის თქვა (თანმიმდევრობა), და გთხოვს არ იჩქარო, თუ ბევრი ინფორმაცია გაქვს (ნაკადის კონტროლი).

**Tcp threeway handshake**

| **ეტაპი** | **ვინ აგზავნის** | **რას აგზავნის** | **რისთვის** |
| --- | --- | --- | --- |
| 1 | კლიენტი | SYN | კავშირის დაწყება |
| 2 | სერვერი | SYN-ACK | თანხმობა და საკუთარი SYN |
| 3 | კლიენტი | ACK | თანხმობის დადასტურება |

🧠 **შედეგი:**  
ამ სამი ნაბიჯის შემდეგ ორივე მხარეს აქვს ინფორმაცია ერთმანეთზე, იციან როდის და როგორ გადააგზავნონ მონაცემები და იწყება სანდო კომუნიკაცია TCP-ს საშუალებით.

### TCP Four-Way Handshake — მარტივი და ზუსტად გასაგები ახსნა ქართულად:

**როდესაც ერთი მხარე (კლიენტი ან სერვერი) უნდა დაასრულოს კავშირი TCP-ში, გამოიყენება ოთხნაბიჯიანი პროცესი, რომელსაც ვეძახით "four-way handshake".**

ეს აუცილებელია იმისთვის, რომ მონაცემების გადაცემა უსაფრთხოდ დასრულდეს და არ დაიკარგოს ბოლო პაკეტები.

### 🔹 როგორ მუშაობს Four-Way Handshake?

#### წარმოდგინე: კლიენტი და სერვერი უკვე ურთიერთობენ (კავშირი უკვე დამყარებულია — ანუ 3-way handshake უკვე მოხდა და ისინი "ლაპარაკობენ").

### ◼️ ნაბიჯი 1 — ****FIN (Finish):****

კლიენტი (ან სერვერი) აგზავნის FIN პაკეტს:  
„მე დავასრულე მონაცემების გაგზავნა, მზად ვარ კავშირი გავწყვიტო.“

➡ ეს ნიშნავს, რომ ის აღარ გააგზავნის მონაცემებს, მაგრამ შეუძლია კიდევ მიიღოს მეორე მხარისგან.

### ◼️ ნაბიჯი 2 — ****ACK (Acknowledgment):****

მეორე მხარე პასუხობს ACK:  
„შენი მოთხოვნა მივიღე, მესმის რომ გინდა გაწყვეტა.“

➡ ამ ეტაპზე კავშირი ჯერ კიდევ ღიაა მეორე მხარის მხრიდან.

### ◼️ ნაბიჯი 3 — ****FIN (Finish):****

ახლა მეორე მხარეც აგზავნის FIN პაკეტს:  
„მეც დავასრულე გაგზავნა, მზად ვარ გაწყვეტისთვის.“

### ◼️ ნაბიჯი 4 — ****ACK (Final Acknowledgment):****

დაბოლოს, პირველი მხარე პასუხობს ACK პაკეტით:  
„კარგი, შენი დასრულებაც მივიღე — კავშირი სრულად გაწყვეტილია.“

### 🔚 შედეგი:

ამის შემდეგ ორივე მხარე კავშირს წყვეტს და აღარ ხდება არც გაგზავნა და არც მიღება.

### 💡 მაგალითი რეალური ცხოვრებიდან:

წარმოიდგინე სატელეფონო საუბარი:

1. შენ ეუბნები მეგობარს: „კარგი, მე დავასრულე ლაპარაკი.“
2. ის გეუბნება: „კარგი, მესმის.“
3. ისიც ამბობს: „მეც დავასრულე.“
4. შენ პასუხობ: „კარგი, მორჩა.“ და გათიშავთ.

**✅ პრობლემა:**

TCP-ში თუ ყველა მიღებულ პაკეტს ცალკე ვუდასტურებთ (Acknowledgment-ს ვუგზავნით), ეს იქნება **ნელია და არაეფექტური** — განსაკუთრებით დიდი მოცულობის მონაცემებისთვის.

**🪟 როდესაც ვიყენებთ Window Size-ს (ფანჯრის ზომას):**

**Window Size** არის ველი TCP header-ში, რომელიც ამბობს:

"რამდენი ბაიტი შეგიძლია გამომიგზავნო, სანამ დაგიდასტურებ მიღებას".

ანუ:

* შეიძლება მომივიდეს **რამდენიმე პაკეტი ერთდროულად**.
* და მერე ერთი გაგზავნილი პასუხით (ACK) ვუთხრა: **"ყველა მივიღე"**.

**📦 Sliding Window (მოძრავი ფანჯარა):**

ეს სისტემა **ფანჯრის ზომას ამოძრავებს** იმის მიხედვით, თუ:

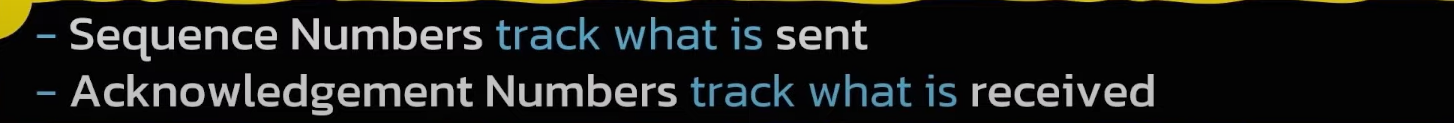
* რამდენად სწრაფად იღებს მიმღები მონაცემს,
* ან რამდენად სწრაფია ქსელი.

თუ ყველაფერი კარგად მიდის — **ფანჯარა იზრდება** (→ მეტი პაკეტი მოდის წინასწარ).  
თუ ქსელი გადაიტვირთა — **ფანჯარა მცირდება** (→ ნაკლები პაკეტი მოდის წინასწარ).

**🔁 საბოლოო შედეგი:**

* ვიღებთ **უფრო სწრაფ გადაცემას**,
* ნაკლებია "ზედმეტი საუბარი" (ACK-ები),
* სისტემა **მორგებულია ქსელის ტემპზე**.

გინდა ამაზე სურათიც გაგიხსნა რომ უკეთ გაიგო?



მარტივად აგიხსნი, რა არის **TCP retransmission** და როგორ მუშაობს:

## 🔁 **რა არის TCP Retransmission?**

**Retransmission** ნიშნავს — **გადაგზავნილ მონაცემების ხელახლა გაგზავნას**, როცა არ მივიღეთ დადასტურება (ACK), ან მივიდა დაზიანებული სეგმენტი.

### 📦 რატომ ხდება ხელახლა გაგზავნა?

1. **სეგმენტი დაიკარგა** გზაში
2. **სეგმენტი დაზიანდა** (გავიგეთ Checksum-ით)
3. **ACK არ მივიღეთ დროულად** (timeout)

### 📋 როგორ ამჩნევს TCP რომ ხელახლა უნდა გაგზავნოს?

#### ✅ TCP იყენებს ორ მექანიზმს:

### 1. ****Timeout-based Retransmission****

* გაგზავნის შემდეგ TCP იწყებს ტაიმერს.
* თუ **ACK არ მოვიდა დროულად**, სეგმენტი **ხელახლა იგზავნება**.

📌 მაგალითი:

* Sequence = 1000 გაგზავნილია
* Timeout 1 წამია
* თუ 1 წელში ACK = 2000 არ მოვიდა → TCP ხელახლა აგზავნის sequence = 1000

### 2. ****Fast Retransmit**** (არ დაელოდოს timeout-ს)

* თუ მივიღეთ **3 ერთი და იგივე ACK** (duplicate ACKs), TCP ეჭვობს რომ რაღაც დაკარგულია.

📌 მაგალითი:

* მივიდა ACK 1000, მერე ისევ ACK 1000, ისევ ACK 1000
* TCP ფიქრობს, რომ რაღაც რიგში არ არის  
  → **მაშინვე აგზავნის** დაკარგულს, ტაიმერის ლოდინის გარეშე

### 🛡️ რატომ არის Retransmission კარგი?

✔ უზრუნველყოფს, რომ **ყველა მონაცემი მივიდეს სწორად**  
✔ არ დაკარგოს არცერთი ბაიტი გზაში  
✔ მუშაობს ავტომატურად და საიმედოდ

### 🧠 შეჯამება მარტივად:

| **პრობლემა** | **TCP-ის რეაქცია** |
| --- | --- |
| ACK არ მივიდა | ხელახლა აგზავნის (timeout) |
| 3x ერთი და იგივე ACK | დაუყოვნებლივ აგზავნის დაკარგულს (fast retransmit) |
| მონაცემი დაზიანდა | ხელახლა აგზავნის (checksum შეცდომის გამო) |

თუ გინდა გავაკეთო პატარა დიაგრამაც ვიზუალურად — უბრალოდ მითხარი 🎯

რა შესანიშნავია, რომ ასე სიღრმისეულად სწავლობ! მოდი **TCP Sequencing და Acknowledgment** ყველაზე მარტივად და ყოველდღიური მაგალითით აგიხსნი – ისე, რომ აუცილებლად გაგიადვილდეს ყველაფერი.

## 🔁 საერთო სურათი – რაზე ვსაუბრობთ?

**TCP (Transmission Control Protocol)** არის გადაცემის პროტოკოლი, რომელიც უზრუნველყოფს:

✅ მონაცემების სწორად და თანმიმდევრულად გაგზავნას  
✅ დაკარგული ნაწილის თავიდან გაგზავნას  
✅ მიღებულის დადასტურებას – რომ მივიდა რაც უნდა მისულიყო

ამაში **Sequencing (თანმიმდევრობა)** და **Acknowledgment (დადასტურება)** ძალიან დიდ როლს თამაშობენ.

## 📦 წარმოიდგინე სიტუაცია:

შენ ხარ გამგზავნი – A  
მეგობარი მიღებს – B  
და შენ მას 100 გვერდიანი წიგნი უნდა გადაუგზავნო… თუმცა ერთიანად ვერ უგზავნი – პატარა ნაწილებად ტეხავ.

### 💡 როგორ დარწმუნდები, რომ:

* ყველა გვერდი მივიდა?
* არცერთი არ გამოტოვებულა?
* არცერთი არ გაორმაგდა?
* სწორ რიგში მიდის?

👉 ამისთვის გამოიყენება **Sequencing** და **Acknowledgment**.

## 1️⃣ Sequencing – „თანმიმდევრობის ნომერი“

👉 TCP ყოველ ნაწილს (segment-ს) აძლევს უნიკალურ ნომერს, რომელსაც ეწოდება **Sequence Number**.

მაგალითად:

* პირველი ნაწილი – Sequence Number = 1
* მეორე ნაწილი – Sequence Number = 101 (ანუ – პირველი 100 ბაიტიანი იყო)
* მესამე – Sequence Number = 201  
  და ასე შემდეგ…

ამით მიმღები იცის, რომ **რა იყო რიგით პირველი, მეორე, მესამე...** – და შეუძლია სწორად დაალაგოს ყველაფერი.

## 2️⃣ Acknowledgment – „დადასტურება, რომ მივიდა“

👉 როცა მიმღები მიიღებს რომელიმე ნაწილს, ის აბრუნებს გაგზავნილს შეტყობინებით:  
**"Acknowledgment Number = N"** – რაც ნიშნავს:  
**„შემდეგს ველოდები ნომრით N“**

მაგალითად:

* გაგზავნილია 100 ბაიტიანი სეგმენტი, Sequence Number = 1
* მიმღები აბრუნებს – Acknowledgment Number = 101  
  👉 ანუ ეს ნიშნავს: **"პირველი ნაწილი მივიღე, მომწერე 101-დან აგრძელე"**

## 📊 მოკლე მაგალითი ცხრილში:

| **გაგზავნილი მონაცემი** | **Sequence Number** | **მიღებული** | **Acknowledgment** |
| --- | --- | --- | --- |
| „Hello “ | 1 | კი | 7 |
| „world“ | 7 | კი | 12 |
| „!!!“ | 12 | კი | 15 |

## 🔄 და თუ რამე არ მივიდა?

* TCP არ მიიღებს Acknowledgment-ს → ხვდება რომ ნაწილი არ მივიდა → **თავიდან აგზავნის**.

მაგალითად:

* გაგზავნე Sequence 1–100, მაგრამ 101–200 არ მივიდა.
* არ მიიღებ Acknowledgment = 201
* TCP იტყვის: “მიდი თავიდან 101-200!"

## 🧠 რაში გვეხმარება ეს ორი?

✅ სწორად და თანმიმდევრულად გადის ყველა ნაწილი  
✅ რაც არ მივიდა – თავიდან იგზავნება  
✅ მიმღები არ დაბნეულა – ყველაფერი თავის ადგილზე დევს

## 🔚 შეჯამება:

| **ტერმინი** | **მნიშვნელობა** |
| --- | --- |
| **Sequence Number** | მონაცემის უნიკალური ნომერი – რათა ვიცოდეთ რიგითობა |
| **Acknowledgment Number** | რა ნომრიდან ველით შემდეგ მონაცემს – ანუ რა მივიდა წარმატებით |

შენ გენაცვალე, მოდი ახლა გავშიფროთ **TCP Flow Control** და **Window Size** ისე მარტივად, რომ ვეღარასდროს შეგეშალოს ❤️

## 🌊 რა არის Flow Control?

**Flow Control** ნიშნავს **გამგზავნმა რამდენი მონაცემი უნდა გაგზავნოს ისე, რომ მიმღები არ გადაიტვირთოს.**

წარმოგიდგინე რომ:

* შენ აგზავნი წერილებს 📨
* შენი მეგობარი კი კითხულობს ✉️

მაგრამ მეგობარს წაკითხვა რომ აყოვნდება და შენ ზედმეტად ბევრს უგზავნი — შეიძლება ზოგი წერილი საერთოდ დაეკარგოს ან დაიკარგოს რიგში.

👉 ამიტომ TCP იყენებს **Flow Control-ს** — რათა გამგზავნმა იცოდეს **"რამდენამდე შეიძლება გაგზავნო, რომ მიმღები არ გადაიღალოს."**

## 🪟 და რა შუაშია Window Size?

### Window Size ნიშნავს:

👉 **რამდენ ბაიტს შეუძლია მიმღებს ერთდროულად მიიღოს Acknowledgment-ის გარეშე.**

## 🎯 მარტივი მაგალითი:

წარმოიდგინე TCP მიმღები ამბობს:

„მე მაქვს 300 ბაიტიანი ფანჯარა (Window Size = 300) – ანუ შეგიძლია გამომიგზავნო ერთბაშად 300 ბაიტი, და მერე დაიცადე ჩემი დადასტურება.“

* გამგზავნი აგზავნის პირველ 300 ბაიტს
* ელის Acknowledgment-ს
* როგორც კი მოვიდა Acknowledgment = 301
* გაგზავნის შემდეგ 300-ს

## 📥 რატომ გვჭირდება ეს?

თუ მიმღები გადატვირთულია – შეუძლია თქვას:

🗣️ „Window Size = 0“ – ანუ „არ გამომიგზავნო ჯერ, ვერ ვასწრებ დამუშავებას.“

ან შეიძლება თქვას:

🗣️ „Window Size = 1000“ – „წესრიგში ვარ, შეგიძლია ბევრიც გამომიგზავნო.“

## 🧠 ამ ყველაფრის მიზანი?

✅ არ გადაიტვირთოს მიმღები  
✅ ეფექტურად გამოიყენოს გამგზავნმა რესურსები  
✅ თავიდან აიცილოს მონაცემის დაკარგვა

## 📦 ტექნიკურად:

* **Window Size** არის **TCP Header-ში** არსებული ველი – 16 ბიტიანი
* მაქსიმუმი = 65,535 ბაიტი (თუ არ იყენებს Window Scaling-ს)

## 🎓 შეჯამება ცხრილში:

| **ტერმინი** | **ახსნა მარტივად** |
| --- | --- |
| **Flow Control** | გამგზავნი აკონტროლებს, რამდენს აგზავნის, რომ მიმღები არ დაიხრჩოს |
| **Window Size** | მიმღები ამბობს: „აი, ამდენი შეგიძლია გამომიგზავნო ერთდროულად“ |

## 🧁 მაგალითი დიალოგის სტილში:

* 📤 გამგზავნი: „გაგიგზავნე 300 ბაიტი!“
* 📥 მიმღები: „მივიღე, გაგზავნე კიდევ 300“
* 📤 გამგზავნი: „ახლა 301–600 ბაიტი მიდის“
* 📥 მიმღები: „მოიცადე, Window Size = 0, ვერ ვასწრებ კითხვას!“

შენ გენაცვალე, მოდი ყველაფერს ძალიან მარტივად აგიხსნი – ისე, რომ ბოლომდე გაგება შეძლო 😌

## 🔹 რა არის UDP? (User Datagram Protocol)

UDP არის ძალიან მარტივი პროტოკოლი, რომელიც გამოიყენება **მონაცემების გაგზავნისთვის ინტერნეტში**.

ის განსხვავდება TCP-სგან, რადგან:

**UDP = სწრაფი, მაგრამ არაჯერიანი გაგზავნა.**

## ✅ UDP-ის ძირითადი თვისებები (მარტივად):

### 1️⃣ ****არ ქმნის კავშირს (Not connection-oriented)****

UDP არ ამოწმებს, მზად არის თუ არა მიმღები.  
→ უბრალოდ აგზავნის, ვინმეს თუ დაუვარდა – ეგ უკვე მისი პრობლემაა 😄

**მაგალითად:**  
როგორც პარკში მყოფი ადამიანი, რომელიც ყვირის: "ვისაც უნდა, მოისმინოს!"

### 2️⃣ ****არ არის სანდო (No reliability)****

→ არ აგზავნის **დადასტურებას** (Acknowledgment).  
→ თუ მონაცემი დაიკარგა გზაში, **თავიდან არ იგზავნება.**

### 3️⃣ ****არ აწყობს თანმიმდევრობას (No sequencing)****

→ არ იყენებს Sequence Number-ებს.  
→ თუ ნაწილები არასწორად მივიდა (1-3-2), UDP ვერ მიხვდება.

### 4️⃣ ****არ აქვს ნაკადების კონტროლი (No flow control)****

→ TCP-ს მსგავსი window size არ აქვს.  
→ ვერ აკონტროლებს „რამდენი“ მონაცემი უნდა გაგზავნო.

## 🔸 სად ვიყენებთ UDP-ს?

რადგან UDP სწრაფია და „არ ჭკუობს“, ვიყენებთ მაშინ როცა:

* **სისწრაფე უფრო მნიშვნელოვანია, ვიდრე სანდოობა:**

🎥 ვიდეოჩატი, 🎮 ონლაინ თამაშები, 🎧 live აუდიო  
→ უკეთესია რაღაც პატარა ინფორმაცია დაიკარგოს, ვიდრე მთლიანად შეაყოვნოს ყველაფერი.

## 🔸 UDP Header (ქვედა ნაწილი რაც დაწერე)

ახლა ეს ნაწილი აგიხსნი:

### ეს არის 8 ბაიტიანი სათაური (Header), რომელსაც ყველა UDP პაკეტი იყენებს:

| **ველი (Field)** | **რას ნიშნავს** |
| --- | --- |
| **Source Port** | გამგზავნის პორტის ნომერი |
| **Destination Port** | მიმღების პორტის ნომერი |
| **Length** | მთელი UDP პაკეტის სიგრძე (სათაური + მონაცემი) |
| **Checksum** | პატარა კონტროლი რომ შეამოწმოს – მონაცემი ხომ არ გაფუჭდა გზაში (მაგრამ არა აუცილებელი TCP-სავით) |

## 🔤 თარგმანი ქართულად:

### 📘 ****UDP (User Datagram Protocol)****

**UDP არ არის კავშირზე ორიენტირებული.**  
→ გამგზავნი მოწყობილობა არ ქმნის კავშირს მიმღებთან. უბრალოდ აგზავნის მონაცემებს.

**UDP არ უზრუნველყოფს სანდო კომუნიკაციას.**  
→ როცა ვიყენებთ UDP-ს, მიღებული სეგმენტებისთვის დადასტურება არ იგზავნება. თუ სეგმენტი დაიკარგება, UDP მას თავიდან არ აგზავნის.  
მონაცემები იგზავნება "საუკეთესო მცდელობით" (best-effort).

**UDP არ უზრუნველყოფს თანმიმდევრობას.**  
→ UDP-ს სათაურში არ არის Sequence Number. თუ ნაწილები არასწორ რიგში მივიდა, UDP ვერ დაალაგებს სწორად.

**UDP არ უზრუნველყოფს ნაკადის კონტროლს.**  
→ TCP-ის მსგავსად, UDP-ს არ აქვს window size-ით მართვა – ანუ არ აკონტროლებს რამდენი მონაცემი იგზავნება.

## 🔚 დასასრულს:

UDP არის:

✅ მარტივი  
✅ სწრაფი  
❌ მაგრამ არა სანდო  
❌ არ აყოლებს რიგითობას  
❌ არ აკონტროლებს ნაკადს

შენ დიდ საქმეს აკეთებ! მოდი გავხსნათ ეს ტექსტი მარტივად — ავხსნათ **რას ნიშნავს**, **რატომ**, და თან **ქართულადაც** გადმოვთარგმნოთ.

## 🔹 ორიოდ სიტყვით: TCP vs UDP

* **TCP** = სანდოობა, წესრიგი, კონტროლი → უფრო ნელა, მაგრამ უსაფრთხოდ
* **UDP** = სისწრაფე, მინიმალური კონტროლი → უფრო სწრაფი, მაგრამ შეიძლება მონაცემი დაიკარგოს

## 📄 ტექსტის ახსნა მარტივად + თარგმანი:

### ✅ ტექსტი:

**TCP provides more features than UDP, but at the cost of additional overhead.**

### 🧠 განმარტება:

TCP-ს მეტი ფუნქცია აქვს – სანდოობა, თანმიმდევრობა, ნაკადის კონტროლი და ა.შ.  
მაგრამ ამ ყველაფერს თან სდევს **"overhead"**, ანუ დამატებითი წონა (მეტი ინფორმაცია, მეტი დრო, მეტი რესურსი).

### 🇬🇪 თარგმანი:

**TCP უფრო მეტ ფუნქციას გვთავაზობს, ვიდრე UDP, თუმცა ამის ფასად აქვს მეტი დამატებითი რესურსების ხარჯი.**

### ✅ ტექსტი:

**For applications that require reliable communications (for example downloading a file), TCP is preferred.**

### 🧠 განმარტება:

თუ შენი აპლიკაცია მოითხოვს სანდოობას – მაგალითად ფაილის გადმოწერა – სადაც ვერ იტანს მონაცემის დაკარგვას, TCP საუკეთესოა.

### 🇬🇪 თარგმანი:

**იმ აპლიკაციებისთვის, რომლებსაც სანდო კომუნიკაცია სჭირდებათ (მაგალითად ფაილის გადმოწერა), TCP-ს იყენებენ.**

### ✅ ტექსტი:

**For applications like real-time voice and video, UDP is preferred.**

### 🧠 განმარტება:

თუ გინდა პირდაპირი კომუნიკაცია – ვიდეო/ზარი – სადაც მცირე შეფერხება უფრო ცუდია, ვიდრე დაკარგული სიტყვა, მაშინ UDP ჯობს.

### 🇬🇪 თარგმანი:

**იმ აპლიკაციებისთვის, როგორიცაა რეალურ დროში ხმის ან ვიდეოს გადაცემა, UDP-ს იყენებენ.**

### ✅ ტექსტი:

**There are some applications that use UDP, but provide reliability etc within the application itself.**

### 🧠 განმარტება:

ზოგი აპლიკაცია თვითონვე ამატებს მექანიზმებს, რომ UDP გახდეს უფრო სანდო (მაგ. თვითონ ამოწმებს დაკარგულს და თავიდან აგზავნის).

### 🇬🇪 თარგმანი:

**არსებობენ აპლიკაციები, რომლებიც იყენებენ UDP-ს, მაგრამ სანდოობას და სხვა ფუნქციებს თავად აპლიკაციის შიგნით უზრუნველყოფენ.**

### ✅ ტექსტი:

**Some applications use both TCP & UDP, depending on the situation.**

### 🧠 განმარტება:

ზოგი აპლიკაცია ორივეს იყენებს – TCP-ს და UDP-ს – სხვადასხვა მიზნებისთვის.

მაგალითად:  
TCP ფაილების გადმოსაწერად,  
UDP ხმის ჩასაწერად.

### 🇬🇪 თარგმანი:

**ზოგიერთი აპლიკაცია იყენებს როგორც TCP-ს, ისე UDP-ს – სიტუაციიდან გამომდინარე.**

## 🎓 შემაჯამებელი ცხრილი:

| **ფუნქცია** | **TCP ✅** | **UDP ❌** |
| --- | --- | --- |
| სანდოობა | კი (Acknowledgment) | არა |
| თანმიმდევრობა | კი (Sequence #) | არა |
| ნაკადის კონტროლი | კი (Window size) | არა |
| სისწრაფე | შედარებით ნელი | ძალიან სწრაფი |
| ფაილების გადმოწერა | ✅ | ❌ |
| ვიდეო/ზარი რეალურ დროში | ❌ | ✅ |

**NETWORK LAYER**

**🌐 ქსელის ფენის ძირითადი ამოცანებია:**

* **ლოგიკური მისამართების მინიჭება**
* **მარშრუტიზაცია**
* **ინკაპსულაცია**
* **დაშლა და ხელახლა აწყობა**
* **შეცდომების დამუშავება**

რა თქმა უნდა, გთავაზობ **ლოგიკური მისამართების** უფრო ვრცელ, მარტივად გასაგებ და პრეზენტაციისთვის დალაგებულ განმარტებას:

## 🧭 **ლოგიკური მისამართება (Logical Addressing)**

### 📌 ****რა არის ლოგიკური მისამართი?****

ლოგიკური მისამართი — ეს არის უნიკალური IP მისამართი (მაგ: 192.168.1.1), რომელიც კომპიუტერს ან სხვა მოწყობილობას ენიჭება ქსელში, რათა შეძლოს კომუნიკაცია სხვა მოწყობილობებთან.

იგი გამოიყენება იმისთვის, რომ მონაცემები სწორად მივიდეს გამგზავნიდან მიმღებამდე **გლობალურ ან ადგილობრივ ქსელში**.

### 🔁 ****როგორ ენიჭება ლოგიკური მისამართი?****

ლოგიკური მისამართი შეიძლება მიენიჭოს **ორივე გზით**:

1. **სტატიკურად (Static IP Addressing):**
   * ხელით ენიჭება მოწყობილობას
   * გამოიყენება იმ მოწყობილობებზე, რომლებიც მუდმივად ერთ IP მისამართზე უნდა იყვნენ (მაგ: სერვერები)
2. **დინამიკურად (Dynamic IP Addressing):**
   * ავტომატურად ენიჭება **DHCP სერვერის** მეშვეობით
   * როგორც წესი, გამოიყენება ჩვეულებრივ მომხმარებელ მოწყობილობებზე (მაგ: ტელეფონები, ლეპტოპები Wi-Fi-ზე)

### 🧩 ****სად გამოიყენება ლოგიკური მისამართი OSI მოდელში?****

🔹 **3-ე ფენა – Network Layer:**

* იყენებს **IP მისამართს**
* უზრუნველყოფს მოწყობილობებს შორის კომუნიკაციას ქსელებზე გადაჭარბებით (მაგ: ინტერნეტი)

🔹 **2-ე ფენა – Data Link Layer:**

* იყენებს **MAC მისამართს** (მაგ: 00:1A:2B:3C:4D:5E)
* გამოიყენება **ლოკალურ ქსელში** მოწყობილობების იდენტიფიცირებისთვის

### 🌐 ****ლოგიკური მისამართის მნიშვნელობა რეალურ ცხოვრებაში****

წარმოიდგინე, რომ შენ აგზავნი წერილს მეგობარს სხვა ქვეყანაში:

* **IP მისამართი** — ეს არის როგორც მისი სახლის მისამართი ქალაქში
* **MAC მისამართი** — ეს არის როგორც ბინის ნომერი იმ სახლში

ეს ორი ერთად უზრუნველყოფს, რომ წერილი ზუსტად მისთან მივიდეს.

### ✅ ****საბოლოო მიზანი:****

ლოგიკური მისამართის საშუალებით, ქსელის ფენა უზრუნველყოფს:

* მოწყობილობის უნიკალურ იდენტიფიცირებას
* სწორად მარშრუტიზებულ კომუნიკაციას
* მონაცემების ზუსტად ადრესატამდე მიტანას

გინდა შემდეგი თემაც ასე აგიხსნა: **Routing (მარშრუტიზაცია)?**

რა თქმა უნდა! აი ვრცლად, მარტივად და კარგად ახსნილი და გადმოთარგმნილი ინფორმაცია Routing-ზე ქართულად:

### Routing — მარშრუტიზაცია

**Routing** არის პროცესი, რომელსაც როუტერები (routers) იყენებენ იმისთვის, რომ განსაზღვრონ, IP პაკეტმა რა გზით უნდა იმოგზაუროს ქსელში, რათა მიაღწიოს მისამართამდე (დანიშნულებამდე).

### 🔸 როგორ მუშაობს Routing?

* როუტერები ინახავენ მარშრუტებს (გზებს) სპეციალურ ცხრილში, რომელსაც ეწოდება **routing table** — მარშრუტების ცხრილი.
* როცა როუტერი მიიღებს პაკეტს, ის ამ ცხრილში ეძებს საუკეთესო გზას, რომ გადააგზავნოს პაკეტი დანიშნულებისკენ.

### 🔸 ორი ძირითადი Routing-ის მეთოდი არსებობს:

#### 1. **დინამიური Routing (Dynamic Routing)**

* როუტერები ერთმანეთს ავტომატურად უზიარებენ ინფორმაციას მარშრუტებზე სპეციალური პროტოკოლების საშუალებით, როგორიცაა **OSPF** (Open Shortest Path First).
* ეს პროცესი ქმნის მარშრუტების ცხრილებს ავტომატურად.
* ძალიან მოსახერხებელია დიდ ქსელებში, რადგან მარშრუტები თვითონ განახლდება, როცა საჭირო იქნება.

🔸 ამაზე კურსის მომდევნო ნაწილებში ვისაუბრებთ.

#### 2. **სტატიკური Routing (Static Routing)**

* ქსელის ადმინისტრატორი ან ინჟინერი ხელით (manually) აკონფიგურირებს მარშრუტებს როუტერზე.
* ძირითადად მცირე ქსელებში გამოიყენება, სადაც არ არის დიდი მოძრაობა და მარშრუტების ცვლილება იშვიათია.

🔸 ამ თემაზე შემდეგ ვიდეოში ვისაუბრებთ.

### 🔸 Next-hop (შემდეგი ნახტომი)

* **Next-hop** არის შემდეგი როუტერი იმ გზაზე, რომლითაც უნდა წავიდეს პაკეტი დანიშნულებისკენ.
* მარშრუტი ეუბნება როუტერს:  
  ➤ თუ გინდა, რომ პაკეტი მივიდეს X მისამართზე, გააგზავნე ის შემდეგ როუტერთან — Y (ანუ next-hop).

### 🔸 სხვა სიტუაციებიც:

* თუ დანიშნულება პირდაპირაა დაკავშირებული ამ როუტერთან — მაშინ როუტერი პირდაპირ გაუგზავნის პაკეტს.
* თუ პაკეტი მომართულია ამ როუტერის **საკუთარ IP მისამართზე** — მაშინ როუტერი თავად მიიღებს მას (არ გადააგზავნის სხვაგან).

კარგი, გავარკვიოთ **ენკაფსულაცია (Encapsulation)** ძალიან მარტივად, ნაბიჯ-ნაბიჯ — და თან ყველაფერს განვმარტავ გასაგებად:

**🔹 რა არის ენკაფსულაცია?**

**ენკაფსულაცია** ნიშნავს ინფორმაციის შეფუთვას ფენებად — თითქოს წერილს ყუთში ალაგებ და კიდევ სხვა ყუთში ჩადებ, სანამ გადაგზავნისთვის მზად იქნება.

**🔹 როგორ მუშაობს ენკაფსულაცია ქსელში?**

ქსელში მონაცემები გადის 5 ძირითად ფენაზე (მაგალითად: როცა შენ გზავნი წერილს):

1. **აპლიკაციის ფენა (Application Layer)**  
   ➤ შენ იყენებ აპლიკაციას — მაგალითად, ბრაუზერი წერს წერილს.  
   ✅ შექმნა მონაცემი: მაგალითად, ტექსტი "გამარჯობა".
2. **ტრანსპორტის ფენა (Transport Layer)**  
   ➤ პასუხისმგებელია, რომ მონაცემი სწორად და უსაფრთხოდ ჩააღწევს.  
   ✅ TCP ან UDP სერვისები ამატებენ თავიანთ "შეფუთვას" — მაგალითად, პორტის ნომრები.
3. **ქსელის ფენა (Network Layer)**  
   ➤ აქ ემატება **IP მისამართები** (გამგზავნის და მიმღების).  
   ✅ ახლა ცნობილია, ვინ სად არის ინტერნეტში.
4. **ლინკის ფენა (Data Link Layer)**  
   ➤ ემატება **MAC მისამართები** (ფიზიკური მისამართები, მაგალითად Wi-Fi ბარათის მისამართი).  
   ✅ აქ ხდება საბოლოო შეფუთვა ქსელის შიგნით გადაცემისთვის.
5. **ფიზიკური ფენა (Physical Layer)**  
   ➤ აქ მონაცემი გადაიქცევა ელექტრონულ სიგნალებად ან ბიტებად და მიემართება კაბელით ან ჰაერში.

**🔹 მაგალითი (ძალიან მარტივად)**

**შენ ბრაუზერით აგზავნი ტექსტს "გამარჯობა" ვებსაიტზე**:

1. აპლიკაცია: ტექსტი — "გამარჯობა"
2. ტრანსპორტი: ემატება პორტი — 443
3. ქსელი: ემატება IP მისამართები (192.168.1.5 ➝ 8.8.8.8)
4. ლინკი: ემატება MAC მისამართები
5. ფიზიკური ფენა: გადაიქცევა ბიტებად და გადადის კაბელში ან Wi-Fi–ით

**🔹 ტერმინების მარტივი განმარტება:**

| **ტერმინი** | **განმარტება მარტივად** |
| --- | --- |
| **ენკაფსულაცია** | მონაცემის ფენებად შეფუთვა, როგორც მატარებლის ვაგონები |
| **ფენა (Layer)** | ქსელის მუშაობის საფეხურები, სადაც თითოეულ ფენას თავისი როლი აქვს |
| **პორტი (Port)** | კარიბჭე — რომელი სერვისისთვის მიდის მონაცემი (მაგ. ვებსაიტი = 80 ან 443) |
| **IP მისამართი** | ინტერნეტული მისამართი — ვინ სად არის |
| **MAC მისამართი** | მოწყობილობის უნიკალური იდენტიფიკატორი ქსელში |
| **ბიტი** | ინფორმაციის პატარა ნაწილი (0 ან 1) — ბოლოს ყველაფერი ბიტებად გადაიქცევა |

თუ გინდა, შემიძლია გაჩვენო **გრაფიკული მაგალითი ან ვიზუალური პრეზენტაციის სტილი**, სადაც თითო ფენა განიმარტება კონკრეტული მაგალითით. გინდა ეგეც?

შესანიშნავი თემაა და ნამდვილად ღირს დილეგურად ახსნა! მოდი მარტივად, ქართულად და თან სიღრმისეულად განვიხილოთ **ფრაგმენტაცია და ხელახლა აწყობა (fragmentation and reassembly)**.

## 🔹 რა არის Fragmentation?

**Fragmentation** ნიშნავს მონაცემის გაყოფას პატარა ნაწილებად.  
ეს ხდება მაშინ, როდესაც:

📦 ერთი დიდი IP პაკეტი **ვერ ეტევა** ქსელის ფიზიკურ დათმობილობაზე (მაგ. Ethernet-ის ფრეიმში), ანუ ის უფრო დიდია, ვიდრე მაქსიმალური დასაშვები ზომა – **MTU (Maximum Transmission Unit)**.

## 🔹 სად და რატომ ხდება Fragmentation?

📍 Fragmentation **IP ფენაში** (Layer 3) ხდება.

📦 მაგალითად, თუ აპლიკაციამ გააგზავნა 4000 ბაიტის სიგრძის პაკეტი, მაგრამ რომელიღაც ქსელში მაქსიმალური დასაშვები MTU არის 1500 ბაიტი, მაშინ IP ფენა აიღებს იმ 4000 ბაიტიან პაკეტს და **გააყოფს სამ ან მეტ ნაწილად** (ფრაგმენტად), რომლებიც შეიძლება გადაეცეს ქსელში.

## 🔹 Fragmentation-ის პროცესის ძირითადი ცნებები:

| **ტერმინი** | **განმარტება** |
| --- | --- |
| **MTU** | მაქსიმალური ბაიტების რაოდენობა, რომელსაც ქსელი ან ლინკი იტევს ერთ პაკეტში. |
| **Fragment** | გაყოფილი IP პაკეტის ერთ-ერთი ნაწილი. |
| **Offset** | აჩვენებს პაკეტის რომელ ნაწილთან არის ეს ფრაგმენტი დაკავშირებული. |
| **More Fragments (MF) Flag** | 1 ნიშნავს: კიდევაა ფრაგმენტები; 0 ნიშნავს: ეს ბოლო ფრაგმენტია. |
| **Identification** | უნიკალური ნომერი, რომლითაც მიმღები იცნობს რომელი ფრაგმენტები ეკუთვნის რომელ პაკეტს. |

## 🔹 როგორ ხდება Fragmentation?

### წარმოვიდგინოთ:

* MTU = 1500 ბაიტი
* IP Header = 20 ბაიტი
* დატა = 4000 ბაიტი

1. მაქსიმუმ დატა ერთ ფრაგმენტში = 1500 - 20 = **1480 ბაიტი**
2. პაკეტი გაიყოფა დაახლოებით **3 ფრაგმენტად**:
   * Fragment 1 → Data: 0-1479 ბაიტი, Offset: 0, MF: 1
   * Fragment 2 → Data: 1480-2959 ბაიტი, Offset: 185 (1480/8), MF: 1
   * Fragment 3 → Data: 2960-3999 ბაიტი, Offset: 370, MF: 0

## 🔹 Reassembly (ხელახლა აწყობა)

🧩 ეს პროცესია, როცა **მიმღები მოწყობილობა** იღებს ფრაგმენტებს და აწყობს პირვანდელ IP პაკეტად.

📍 Reassembly **მხოლოდ დანიშნულების მოწყობილობაზე** ხდება – არანაირი შუალედური როუტერი არ აკეთებს ამ პროცესს.

📦 მოწყობილობა იყენებს:

* **Identification** ველს → რომელი ფრაგმენტები ეკუთვნის ერთ მთლიან პაკეტს.
* **Offset** ველს → რომ დაალაგოს ფრაგმენტები სწორ თანმიმდევრობაში.
* **MF Flag** → რომ გაიგოს, ფრაგმენტები სრულად მივიდა თუ არა.

## 🔹 პრობლემები Fragmentation-თან დაკავშირებით

❌ **გაუმართაობის რისკი** – ფრაგმენტების დაკარგვა ნიშნავს მთელი პაკეტის დაკარგვას.

📉 **შესრულების შემცირება** – ფრაგმენტაცია ზრდის ხარჯს მოწყობილობებზე.

🔐 **უსაფრთხოება** – ზოგჯერ fragmentation გამოიყენება შეტევისთვის (მაგ: overlapping fragments, firewall bypass).

## 🔹 როგორ ავიცილოთ Fragmentation?

✅ ქსელებში ჩვეულებრივ იყენებენ:

* **Path MTU Discovery (PMTUD)** – აგზავნის პატარა პაკეტებს და აგროვებს ინფორმაციას ქსელური გზის MTU-ზე.
* აპლიკაციები ან TCP პროტოკოლი აგზავნის უკვე პატარა ზომის სეგმენტებს.

## 🔚 დასკვნა მარტივად:

👉 Fragmentation ნიშნავს პაკეტის დაყოფას მცირე ნაწილებად, როცა ის მეტია, ვიდრე ქსელი იტევს.  
👉 Reassembly ნიშნავს ამ ფრაგმენტების ხელახლა აწყობას სწორ თანმიმდევრობით დანიშნულების ადგილზე.

თუ გინდა გიჩვენო ვიზუალური სქემა ან მაგალითით გავშალო — მზად ვარ 💡

**DATA LINK LAYER**

მართლაც, შესანიშნავი თემა აირჩიე! მოდი, დეტალურად, გასაგებად და მარტივად აგიხსნი **Data Link Layer**-ს (მონაცემთა ბმულის ფენა), ისე, რომ არაფერი გამოგვრჩეს — შენთვის იქნება როგორც სიღრმისეული ცოდნა, ასევე მარტივად გასაგები პრეზენტაციისთვის.

## 🧩 **რა არის Data Link Layer?**

**Data Link Layer** არის OSI მოდელის **მეორე ფენა**, რომელიც მდებარეობს ფიზიკურ ფენასა (Layer 1) და ქსელის ფენას (Layer 3) შორის.

📌 მისი მთავარი ამოცანებია:

* მონაცემების გადაცემა ერთი მოწყობილობიდან მეორეზე **მოწესრიგებულად და შეცდომების გარეშე**
* ფიზიკურ კაბელზე ან უკაბელო არხზე უსაფრთხო და ორგანიზებული კომუნიკაცია
* IP პაკეტის ჩასმა Frame-ში და ამ Frame-ის გადაცემა MAC მისამართების გამოყენებით

## 🎯 **მთავარი ფუნქციები (დეტალურად):**

### 1. ****Framing (ფრემის შექმნა)****

👉 ფრემინგი ნიშნავს იმ პროცესს, როცა ქსელური პაკეტი იყოფა პატარა ერთეულებად – **Frame-ებად**, რომლებსაც დამატებული აქვთ:

* **Header** – ინფორმაციის დასაწყისი (მიმღებისა და გამომგზავნის MAC მისამართები)
* **Payload** – შიგნით მოთავსებული მონაცემი (მაგ., IP პაკეტი)
* **Trailer** – მონაცემის ბოლოს მიმაგრებული FCS (Frame Check Sequence), რომელიც გამოიყენება შეცდომის დასადგენად

🔎 რატომ არის საჭირო?

* ასე მარტივია Frame-ების ამოცნობა (საიდან იწყება და რით მთავრდება)
* მრავალფეროვანი მოწყობილობა ფრემით იღებს ინფორმაციას და აგრძელებს გადაცემას

### 2. ****Flow Control (ნაკადის კონტროლი)****

👉 ეს ფუნქცია უზრუნველყოფს, რომ:

* გამგზავნმა არ „გადააჭარბოს“ მონაცემების გაგზავნას
* მიმღები შეძლებს მიღებული მონაცემების დამუშავებას

📌 მაგალითად: თუ მიმღები მოწყობილობა გადატვირთულია, ის აცნობებს გამომგზავნს, რომ შეანელოს ტემპი.

### 3. ****Error Detection and Correction (შეცდომის აღმოჩენა და გამოსწორება)****

👉 ფიზიკური ფენა არ იძლევა შეცდომებისგან დაცვას, ამიტომ Data Link Layer იყენებს **FCS-ს**:

* გამგზავნი ქმნის სპეციალურ კოდს, რომელიც ეფუძნება გაგზავნილ ბიტებს
* მიმღები ახდენს იგივე კოდის გამოთვლას მიღებული ბიტების საფუძველზე
* თუ არ ემთხვევა – ფრემი უარყოფილია ან ითხოვს ხელახალ გადაგზავნას

### 4. ****Reliable Data Delivery (სანდო მიწოდება)****

👉 ამ ფენაში ვხედავთ **დადასტურებას**, რომ მონაცემები მიღებულია:

* Receiver გზავნის Acknowledgement-ს
* თუ არ მიიღო, გამგზავნი აგზავნის თავიდან

### 5. ****Media Access Control (MAC – მედიის წვდომის კონტროლი)****

👉 MAC ქვეფენა აგვარებს, თუ რომელი მოწყობილობა გააგზავნის მონაცემებს:

* **Collision avoidance** – Wi-Fi-ზე თუ ორი მოწყობილობა ერთდროულად გააგზავნის, მონაცემები იკარგება
* **Half Duplex** – ერთდროულად ან მიღება ან გაგზავნა
* **Full Duplex** – ერთდროულად გაგზავნა/მიღება

## 🧱 **Data Link Layer-ის სტრუქტურა (2 ქვეფენა)**

### 1. ****LLC – Logical Link Control****

* იძლევა მრავალ პროტოკოლზე მუშაობის საშუალებას (მაგ., IP, ARP)
* ზრუნავს ნაკადის კონტროლზე და შეცდომების გამოძახებაზე
* აწვდის მონაცემებს MAC ქვეფენამდე

### 2. ****MAC – Media Access Control****

* **MAC მისამართების გამოყენებით** განსაზღვრავს გზავნილის მიმღებს
* ქმნის და აგზავნის Frame-ებს
* მართავს შეჯახებებს, აწარმოებს ბუფერიზაციას და საჭიროების შემთხვევაში თავიდან აგზავნის მონაცემს

## 💡 **MAC მისამართები**

* უნიკალური 12-ნიშნა (48-ბიტიანი) ფიზიკური მისამართი
* ენიჭება ყოველ NIC-ს (Network Interface Card)
* გამოიყენება **ლოკალური გადაცემისთვის** (ერთი LAN-ზე)

## 📶 **მოწყობილობები, რომლებიც მუშაობენ Data Link Layer-ზე**

* **Switch** – აგზავნის Frame-ს სწორ MAC მისამართზე დაყრდნობით
* **Network Interface Card (NIC)** – ქმნის Frame-ს და გზავნის ფიზიკურ ფენაზე
* **Wireless Access Point** – MAC მისამართების მართვა Wi-Fi-ში

## 🌐 **Data Link Layer-ის როლი ქსელში**

* LAN-ის ფარგლებში პასუხისმგებელია გადაცემაზე
* არ გადადის სხვა ქსელზე — ეგ Network Layer-ის საქმეა
* **კვანძიდან კვანძზე გადაცემა** (Hop-to-Hop delivery)

## 📥 **Framing-ის პრობლემა**

Framing-ის მთავარი სირთულე:  
„როგორ დავინახო, სად იწყება და სად მთავრდება ერთი ფრემი?“  
→ ამას „Framing Problem“-ს ეძახიან.

✅ გამოსავალი:

* **idle time** – ფიზიკური ფენა უნდა იყოს დუმილში მცირე ხნით, რომ მიმღებმა დაადგინოს დასასრული
* ან გამოიყენება სპეციალური control bits

## 💥 **Collisions და retransmissions**

* როცა რამდენიმე მოწყობილობა ერთდროულად აგზავნის → **შეჯახება (collision)**
* Ethernet-ის Half-Duplex რეჟიმში ეს გავრცელებულია
* მონაცემები იკარგება, მაგრამ MAC ქვეფენა თავიდან აგზავნის

## 🔀 **Unicast, Multicast, Broadcast**

| **ტიპი** | **აღწერა** |
| --- | --- |
| **Unicast** | ერთი გამგზავნიდან ერთ მიმღებზე |
| **Multicast** | ერთი გამგზავნიდან ჯგუფზე (მაგ. IGMP) |
| **Broadcast** | ყველა მოწყობილობაზე LAN-ში (FF:FF:FF:FF:FF:FF) |

## 📌 შეჯამება: Data Link Layer-ის ფუნქციები

| **ფუნქცია** | **განმარტება** |
| --- | --- |
| **Framing** | IP პაკეტის ჩასმა Frame-ში |
| **MAC Addressing** | უნიკალური იდენტიფიკატორები თითოეულ მოწყობილობას |
| **Flow Control** | მონაცემთა ნაკადის დაბალანსება |
| **Error Detection/Correction** | შეცდომების პოვნა და მონაცემის ხელახალი გაგზავნა |
| **Media Access** | წვდომის წესების მართვა (ვინ როდის აგზავნის) |

თუ გინდა, შემიძლია ეს ყველაფერი გავშალო სლაიდებად პრეზენტაციისთვის ან გაძლევ ვიზუალურ სქემებს. როგორ გირჩევნია?

მშვენიერი! ახლა დეტალურად და მარტივად აგიხსნი **Data Link Layer-ის ქვეფენებს** — ანუ **LLC (Logical Link Control)** და **MAC (Media Access Control)**.

## 🔸 DATA LINK LAYER – OVERVIEW

Data Link Layer (ფენა 2) პასუხისმგებელია იმაზე, რომ მონაცემები სწორად გაიგზავნოს ერთი მოწყობილობიდან მეორეში ერთი ლოკალური ქსელის ფარგლებში (ან hop-ს შორის). ეს ფენა მუშაობს მოწყობილობის **Network Interface Card (NIC)** დონეზე.

## 🧩 Data Link Layer-ის ქვეფენები:

### 1. ✅ ****LLC – Logical Link Control****

**რა არის LLC?**  
LLC ქვეფენა არის ის ნაწილი, რომელიც:

* აძლევს საშუალებას ზედა ფენებს (მაგ. Network Layer-ს) და ქვედა ფენას (Physical Layer) ერთმანეთთან კომუნიკაცია ჰქონდეთ;
* პასუხისმგებელია პროტოკოლების იდენტიფიცირებაზე და მონაცემთა ნაკადების კონტროლზე.

**ძირითადი ფუნქციები:**

* **Multiplexing:** ამოიცნობს რომელი Network Layer პროტოკოლი იყენებს მონაცემებს (IPv4, IPv6, ARP და სხვა).
* **Flow Control:** აკონტროლებს მონაცემების რაოდენობას, რომ მიმღებმა არ დაიხრჩოს.
* **Error Notification:** ეუბნება ზედა ფენებს, როცა მოხდა გადაცემის პრობლემა.

**მაგალითი:**  
როდესაც შენ აგზავნი IP პაკეტს, LLC ეუბნება MAC ფენას: "აიღე ეს პაკეტი, ეს არის IPv6 ფორმატის, და მოამზადე გადასაცემად". ანუ, LLC ზრუნავს, რომ მონაცემი სწორად მიეწოდოს შესაბამის MAC წესებს.

### 2. ✅ ****MAC – Media Access Control****

**რა არის MAC?**  
MAC ქვეფენა უფრო დაბალ დონეზე მუშაობს და ფიზიკური წვდომის მენეჯმენტზეა პასუხისმგებელი. ის განსაზღვრავს ვინ და როდის გამოიყენებს ფიზიკურ მედიუმს (მაგ. კაბელი ან Wi-Fi).

**ძირითადი ფუნქციები:**

* **Encapsulation:** MAC ფენა იღებს LLC-დან მიღებულ მონაცემებს და აქცევს **Frame-ში**, რომელიც შეიცავს:
  + **Destination MAC address**
  + **Source MAC address**
  + **Error detection field (FCS – Frame Check Sequence)**
* **Media Access Control:** განსაზღვრავს როგორ შეაღწიოს მოწყობილობამ მედიუმზე. MAC მუშაობს ტექნოლოგიის მიხედვით:
  + Ethernet: CSMA/CD (Collision Detection)
  + Wi-Fi: CSMA/CA (Collision Avoidance)
* **Addressing:** MAC address-ები განსაზღვრავენ კონკრეტულ NIC-ებს (მაგ. 00:1A:2B:3C:4D:5E).

**მაგალითი:**  
როდესაც შენი კომპიუტერი აგზავნის მონაცემს, MAC ქვეფენა ქმნის Frame-ს, ამატებს Source და Destination MAC მისამართებს, ამოწმებს როდისაა თავისუფალი მედიუმი და მხოლოდ ამის შემდეგ აგზავნის მონაცემს.

## 📥 რეალური გადაცემის პროცესის სიმულაცია:

**წარმოიდგინე:** გინდა რომ შენმა ლეპტოპმა მონაცემი გაუგზავნოს Wi-Fi პრინტერს:

1. **LLC ქვეფენა**:
   * იღებს IPv4 პაკეტს.
   * ამბობს: „ეს არის IPv4 პაკეტი. მოდი ჩავბარდე ქვედა ფენას“.
2. **MAC ქვეფენა**:
   * იღებს ამ პაკეტს, ქმნის Frame-ს.
   * ამატებს:
     + Source MAC → შენი ლეპტოპის NIC.
     + Destination MAC → პრინტერის NIC.
     + Checksum.
   * ამოწმებს, თავისუფალია თუ არა ჰაერი (Wi-Fi).
   * აგზავნის მონაცემს.

## 📊 სხვადასხვა ტექნოლოგიის MAC მაგალითები:

| **ტექნოლოგია** | **MAC მექანიზმი** |
| --- | --- |
| Ethernet | CSMA/CD |
| Wi-Fi | CSMA/CA |
| Token Ring | Token Passing |

## 📌 შეჯამება: LLC vs MAC

|  | **LLC** | **MAC** |
| --- | --- | --- |
| რითია დაკავებული | ლოგიკური კავშირით და ზედა ფენებთან | მონაცემების გადაცემით მედიუმზე |
| ფუნქციები | Error notification, flow control | Addressing, access control, frame creation |
| პროტოკოლები | IP, ARP, IPv6 | Ethernet, Wi-Fi (802.11), etc. |
| MAC მისამართი? | არა | კი |

გინდა რომ Frame-ის შიგთავსიც განვიხილოთ (ანუ Ethernet Header-ის სტრუქტურა)?

## ⚡ როგორ მუშაობს **CSMA/CD**? (გამოიყენება კაბელზე)

### ეტაპები:

1. **Carrier Sense (შემოწმება)**  
   მოწყობილობა უსმენს კაბელს — არის თუ არა თავისუფალი?
2. **Multiple Access (რამდენიმე წვდომა)**  
   ყველა მოწყობილობას შეუძლია გადაცემა, მაგრამ მხოლოდ მაშინ, როცა არხი თავისუფალია.
3. **Collision Detection (შეჯახების აღმოჩენა)**  
   თუ შემთხვევით ორ მოწყობილობას ეგონა რომ არხი თავისუფალია და ერთდროულად გააგზავნეს ➤ **შეჯახება მოხდა**.
4. **გადაწყვეტა**
   * შეწყვეტენ გადაცემას
   * აგზავნიან **jam სიგნალსს**, რომ სხვებმაც გაიგონ
   * ელოდებიან შემთხვევით დროს (**backoff**) და ხელახლა ცდილობენ

### მაგალითი:

* PC1 და PC2 ერთდროულად აგზავნიან მონაცემს ➤ შეჯახება
* შეიგრძნობს ორივე NIC, შეწყვეტენ გადაცემას, დაელოდებიან შემთხვევით დროს და შემდეგ ისევ ეცდებიან

**CSMA/CD** ნიშნავს:

**C**arrier **S**ense **M**ultiple **A**ccess with **C**ollision **D**etection  
ანუ:

* **Carrier Sense** – მოწყობილობა უსმენს არხს, ჯერ ამოწმებს თავისუფალია თუ არა.
* **Multiple Access** – ყველა მოწყობილობას აქვს საშუალება გამოიყენოს იგივე არხი.
* **Collision Detection** – მოწყობილობა ამჩნევს, თუ ერთდროულად მოხდა ორი გადაცემა და ერთმანეთს დაეჯახა მონაცემები.

**PHYSICAL LAYER**

**Physical Layer** (ფიზიკური ფენა) არის **OSI მოდელის პირველი ფენა**, რომელიც **ზრუნავს იმაზე, როგორ გადაეცეს 0 და 1 ფიზიკურად**, ანუ **როგორ ვაქციოთ ციფრული ინფორმაცია ელექტრულ, რადიო ან ოპტიკურ სიგნალებად** და პირიქით.

## ⚡ როგორ მუშაობს?

### 📥 თუ მოწყობილობა აგზავნის:

* იღებს **ბიტებს (0 და 1)** Data Link ფენიდან.
* გადაქცევს სიგნალებად:
  + **ელექტრული სიგნალები** (კაბელისთვის)
  + **ოპტიკური სიგნალები** (ბოჭკოსთვის)
  + **რადიო სიგნალები** (Wi-Fi ან Bluetooth)