Aula 7 - DNS, Programação com Sockets

Diego Passos

Universidade Federal Fluminense

Redes de Computadores I

Material adaptado a partir dos slides originais de J.F Kurose and K.W. Ross.

DNS

DNS: *Domain Name System*

- **Pessoas:** muitos identificadores.
  - CPF, RG, # de passaporte, ...
- **Hosts e roteadores na Internet:**
  - Endereço IP (# de 32 bits) usado para endereçar datagramas.
  - "Nome", *e.g.*, `www.yahoo.com`, usado por humanos.
- **Pergunta:** como mapear nomes para seus respectivos IPs e vice-versa?

- **Domain Name System (DNS):**
  - **Base de dados distribuída:** implementada em uma hierarquia de múltiplos **servidores de nomes**.
  - **Protocolo da camada de aplicação:** *hosts* e servidores de nome se comunicam para **resolver** nomes (tradução entre nome e endereço IP).
    - Note: função fundamental da Internet implementada como protocolo de aplicação.
    - Complexidade nas bordas.

DNS: Serviços, Estrutura

- **Serviços do DNS**:
  - Tradução de nomes de *hosts* para endereços IP.
  - *Host aliasing*.
    - Atribuição de “apelidos”.
    - *Host* possui **nome canônico** e, possivelmente, vários apelidos.
  - *Aliasing* de servidores de e-mail.
  - Balanceamento de carga.
    - Servidores web replicados.
    - Cada servidor com seu IP.
    - Mas o mesmo nome associado a vários IPs.

- **Por que não um DNS centralizado?**
  - Ponto único de falha.
  - Concentração de grande volume de tráfego.
  - Base de dados centralizada distante.
  - Manutenção.

Em suma: **não escala!**

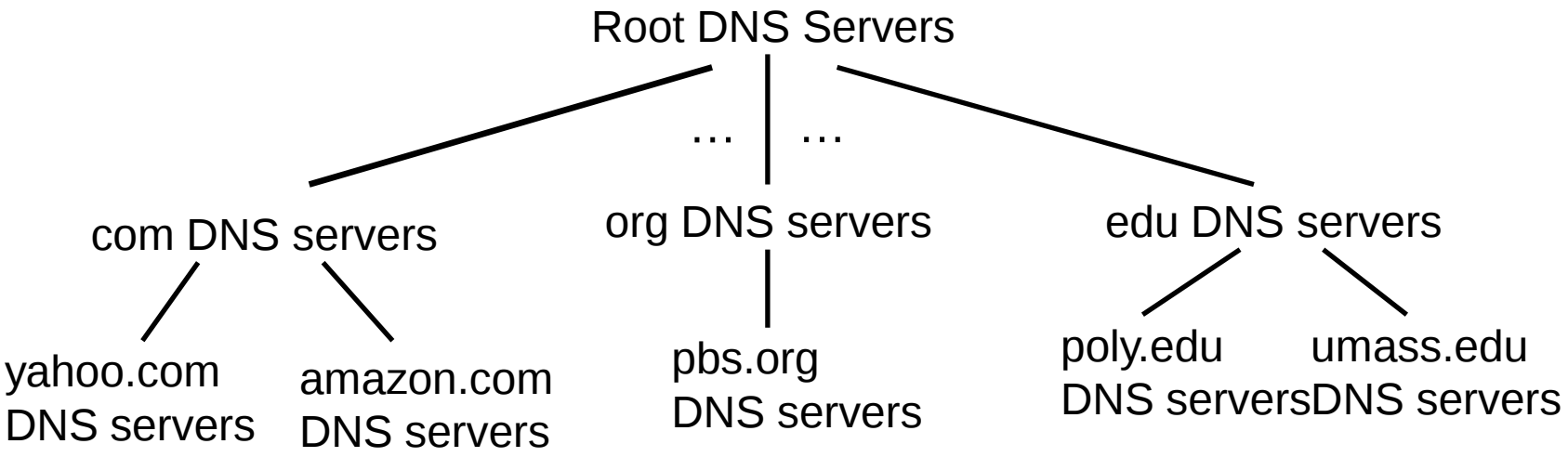
DNS: Balanceamento de Carga

- Aplicação (ping) requisita resolução de nome `www.google.com`:
  - Primeiras execuções associam nome a `216.58.222.100`.
  - Eventualmente, respostas diferentes: `216.58.222.68`.

Alias (Apelidos)

- Primeira resolução para o nome `www.midiacom.uff.br`:
  - Nome associado ao IP `200.20.10.93`.
- Segunda resolução para `mesbla.midiacom.uff.br`:
  - Associado ao mesmo IP!.

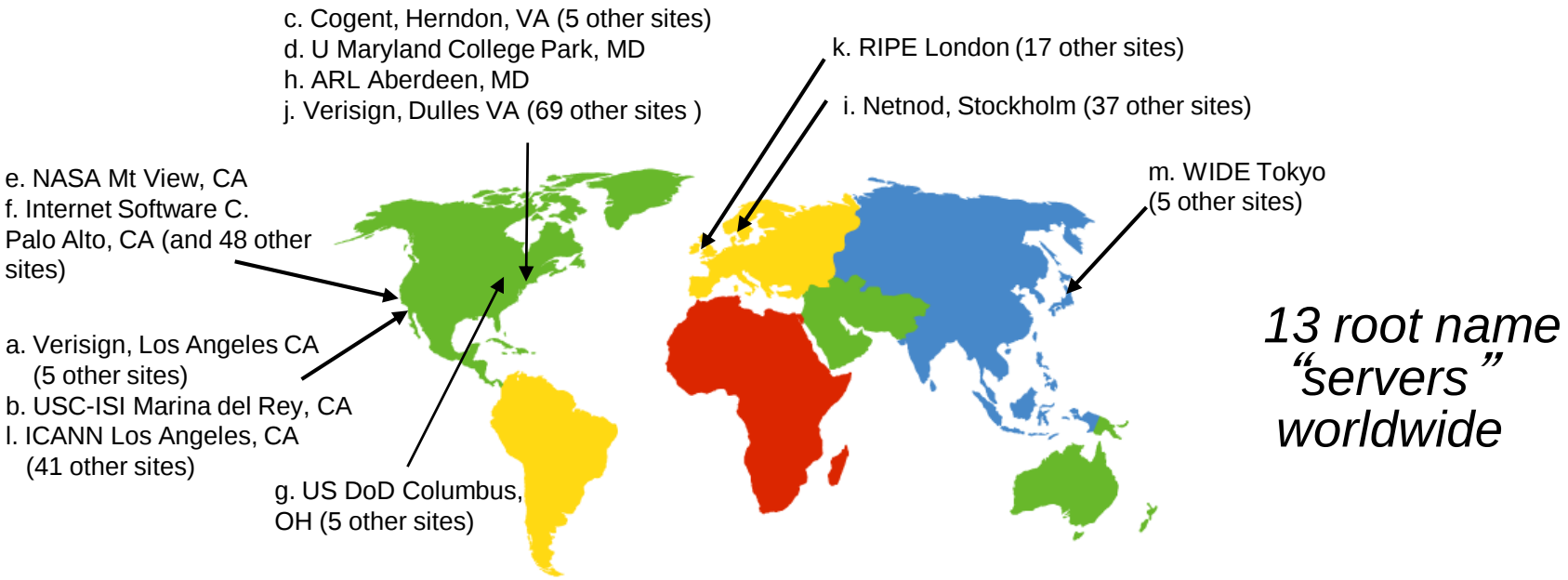
DNS: Uma Base de Dados Hierárquica e Distribuída



- **Cliente quer IP de `www.amazon.com`. Primeira abordagem:**
  - Cliente pergunta ao servidor raiz a localização do DNS do **domínio** `.com`.
  - Cliete pergunta ao servidor DNS do domínio `.com` a localização do servidor DNS do domínio `amazon.com`.
  - Cliete pergunta ao servidor DNS do domínio `amazon.com` pelo endereço IP do *host* `www.amazon.com`.

DNS: Servidores Raiz

- Conhecem os servidores TLD.
- Contactados (principalmente) quando se deseja saber o DNS de um TLD.
- Poucos “servidores” no mundo.
  - Embora cada um seja composto por vários computadores espalhados pelo mundo.



13 root name “servers” worldwide

TLD, Servidores Autoritativos

- **Servidores *Top-Level Domain* (TLD):**
  - TLD: `.org`, `.net`, `.com`, `.edu`, ..., `.br`, `.uk`, `.jp`, ...
  - Cada TLD tem seu servidor DNS específico.
  - A Network Solutions mantém servidores DNS para o TLD `.com`.
  - A Registro.br mantém o DNS para o TLD `.br`.
- **Servidores autoritativos:**
  - Servidor de DNS de uma organização específica.
  - Provê mapeamentos para os endereços IP da organização e seus nomes de *host*.
  - Pode ser gerenciado pela própria organização ou por um provedor de serviço.

TLDs

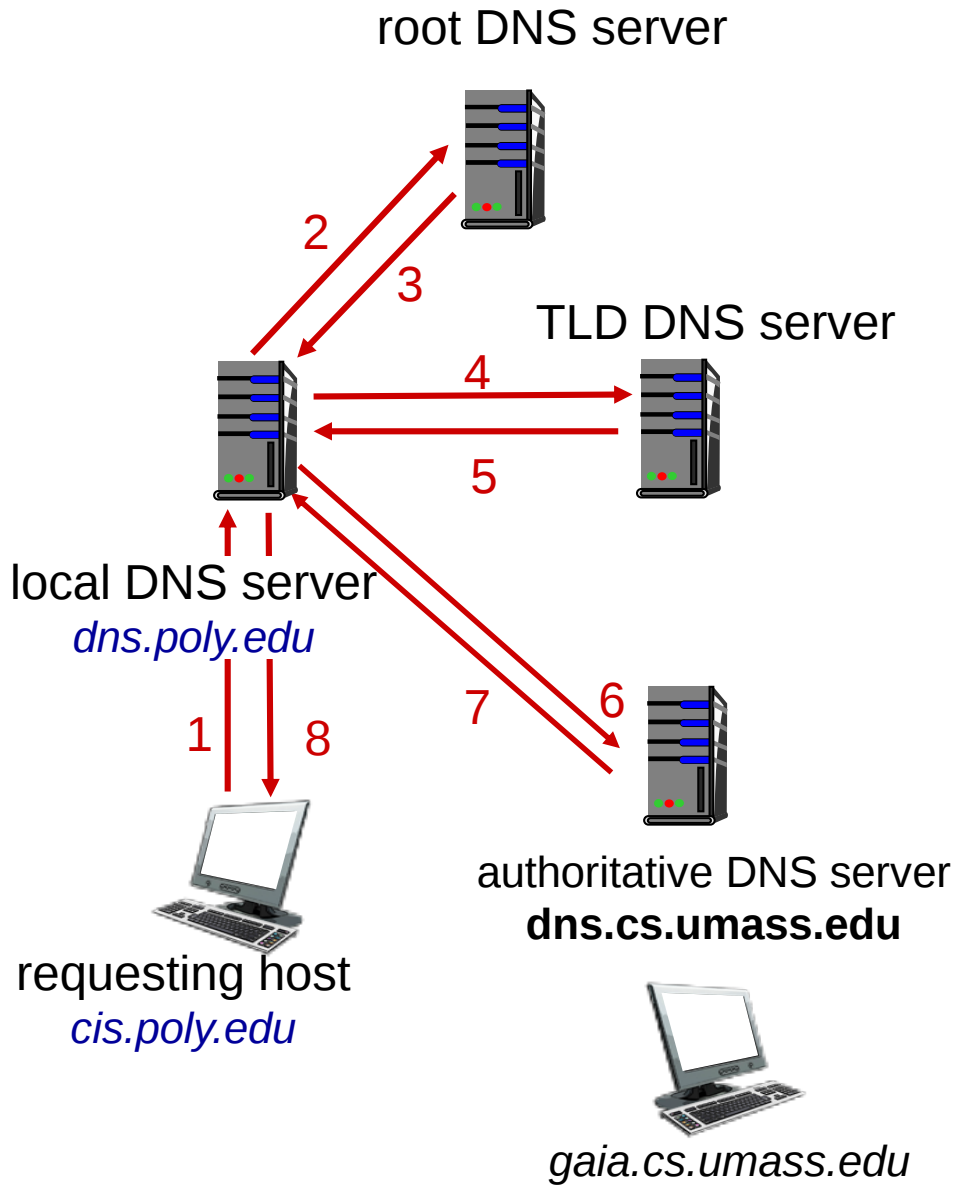
- Primeira resolução para o nome `www.google.com`:
  - Como esperado, bem sucedido.
  - TLD `.com`
- Segunda resolução para `com.google`:
  - Nome parece invertido: primeiro TLD, depois nome da organização.
  - Mas resolução é bem sucedida de qualquer forma!
  - Por que?
    - Atualmente, `.google` é um TLD.
    - Dentro deste TLD, há um *hostname* chamado `com`.

Servidores DNS Locais

- Estritamente falando, não fazem parte da hierarquia.
- Cada ISP (residencial, empresas, universidades) normalmente tem um.
  - Também chamado de "DNS padrão".
- Quando o *host* faz uma requisição DNS, esta é enviada para o seu servidor DNS local.
  - Geralmente, possui um cache para mapeamentos realizados recentemente (mas o mapeamento pode não ser mais válido!).
  - Atua como um *proxy*, encaminhando requisições para a hierarquia.

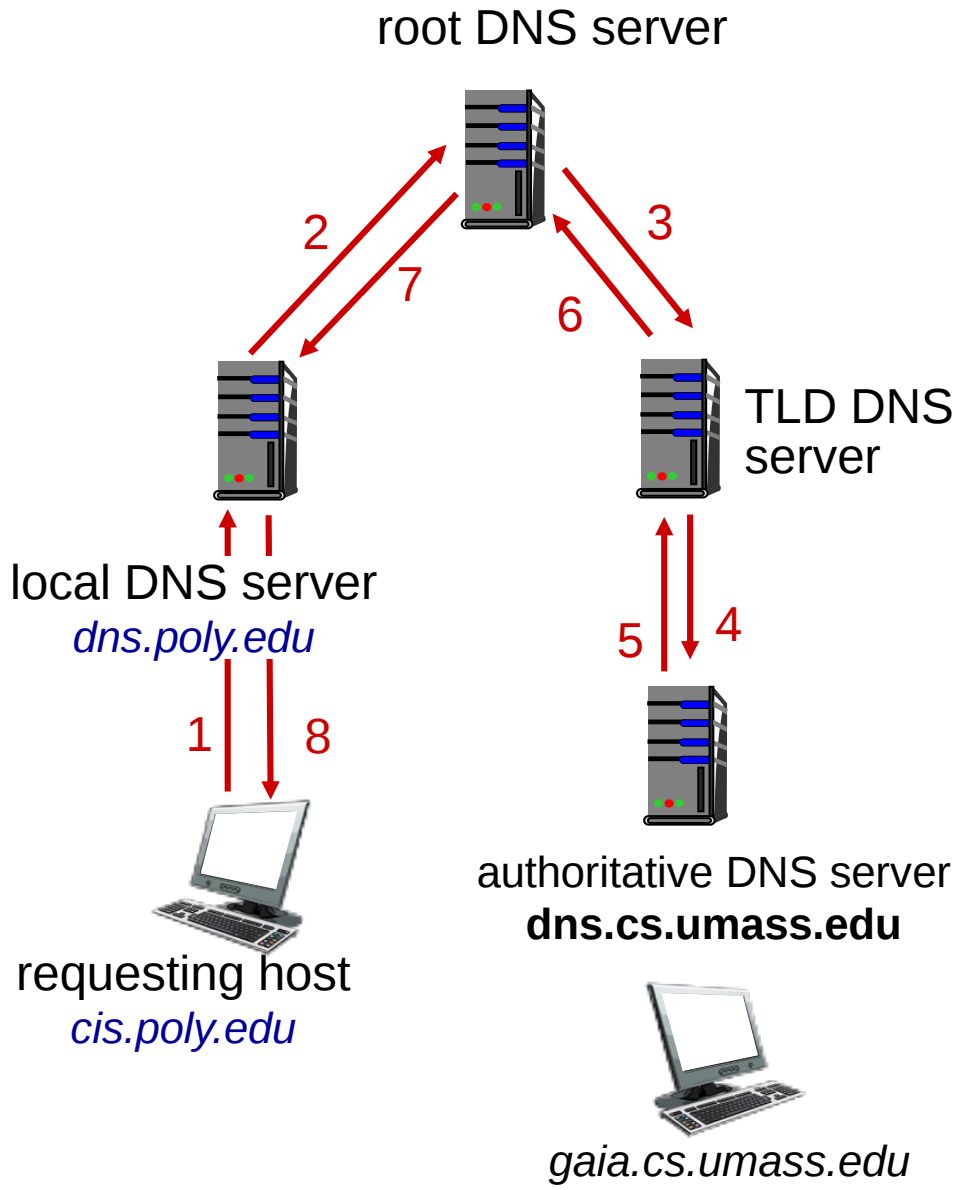
Exemplo de Resolução de Nome Usando DNS (I)

- *Host* em `cis.poly.edu` quer o IP de `gaia.cs.umass.edu`.
- **Método iterativo:**
  - Servidor contactado retorna nome de outro servidor a ser contactado.
  - "Eu não conheço este nome, mas pergunte para este outro servidor".



Exemplo de Resolução de Nome Usando DNS (II)

- **Modo recursivo:**
  - Coloca o fardo da resolução do nome no servidor contactado.
  - Alta carga nos níveis mais altos da hierarquia?



DNS: Cache, Atualização de Registros

- Uma vez aprendido um mapeamento, um servidor de nomes (qualquer) **o armazena em cache.**
  - Entradas na cache tem uma data de expiração (TTL).
    - *i.e.*, são jogadas fora depois de algum tempo.
  - Servidores TLD tipicamente presentes na cache.
    - Logo, servidores raiz raramente visitados.
- Entradas na cache podem ficar **desatualizadas**.
  - Serviço de tradução de melhor esforço!
  - Se *host* tem IP alterado, restante da Internet pode não ficar sabendo até que TTLs expirem.
- Há propostas para mecanismos de atualização/notificação.
  - *e.g.*, RFC 2136.

Registros de DNS

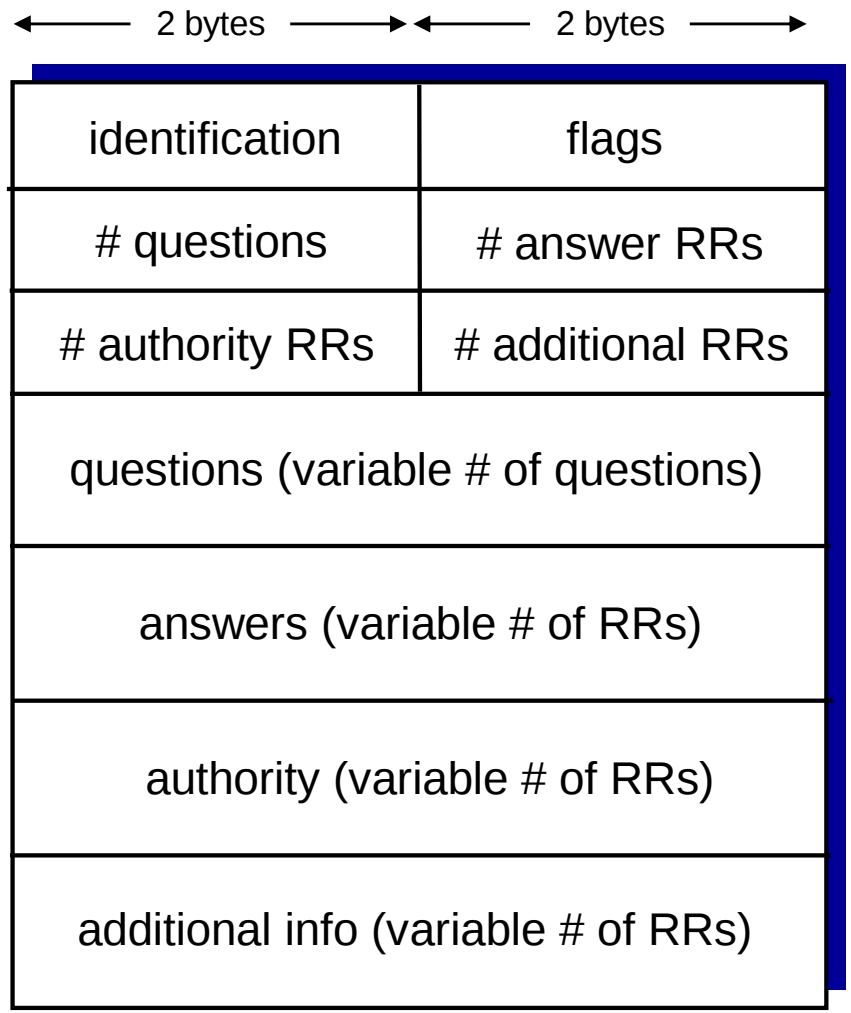
- **DNS:** base de dados distribuída que armazena *Resource Records* **(RR)**.

Formato de um RR: (nome, valor, tipo, TTL)

- **Tipo=A**
  - **nome** é um nome de um *host*.
  - **valor** é o endereço IP.
- **Tipo=NS**
  - **nome** é um domínio (*e.g.*, `foo.com`).
  - **valor** é o **nome do host** do servidor DNS autoritativo para este domínio.
- **Tipo=CNAME**
  - **nome** é um apelido para um *host*.
  - **value** é o **nome canônico**.
  - *e.g.*, `www.midiacom.uff.br` é um apelido para `mesbla.midiacom.uff.br`.
- **Tipo=MX**
  - **valor** é o **nome do host** que funciona como servidor de e-mail do domínio associado ao **nome**.

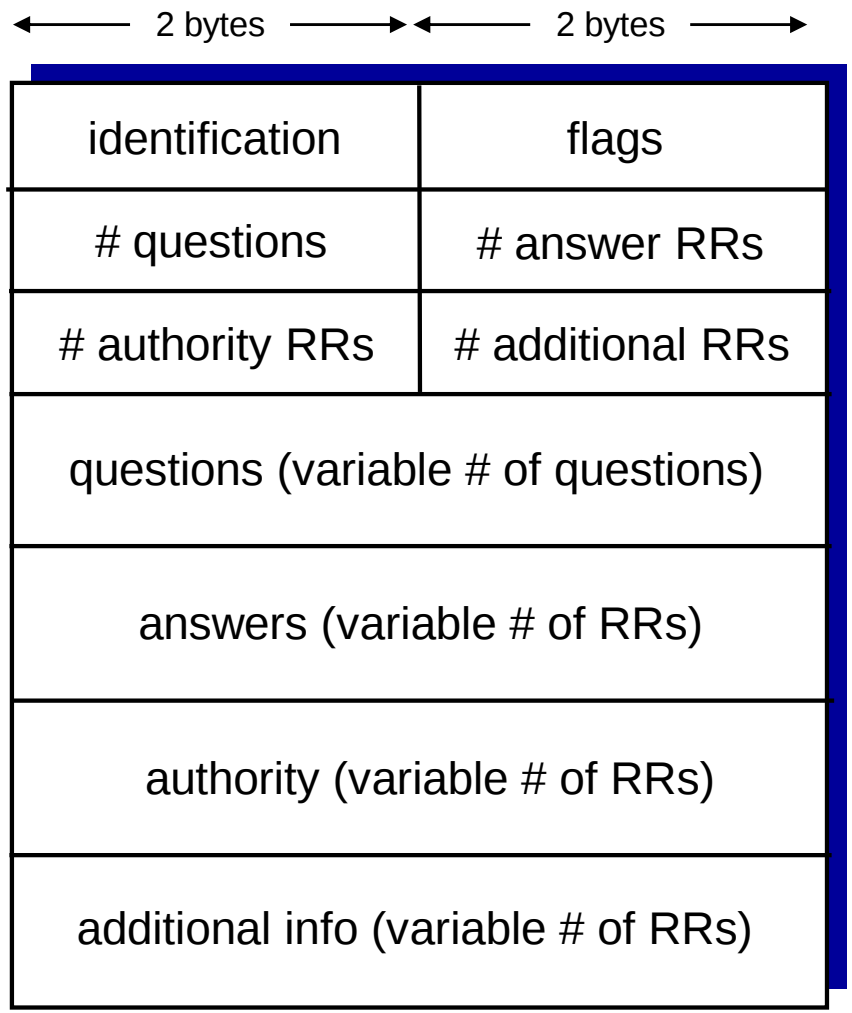
DNS: Protocolo e Mensagens (I)

- Mensagens de **requisição** e **resposta** têm o mesmo **formato**.
- Cabeçalho das mensagens:
  - **Identificação**: # de 16 bits para requisição; resposta utiliza mesmo # da requisição a que responde.
  - **Flags**:
    - Requisição ou resposta.
    - Modo recursivo é desejado.
    - Modo recursivo está disponível.
    - Resposta é autoritativa.



DNS: Protocolo e Mensagens (II)

- Campo das consultas:
  - Múltiplas consultas possíveis em uma mesma requisição.
  - Informa nomes, tipos dos campos nas requisições.
- Campo das respostas:
  - Múltiplas respostas possíveis em uma mesma mensagem.



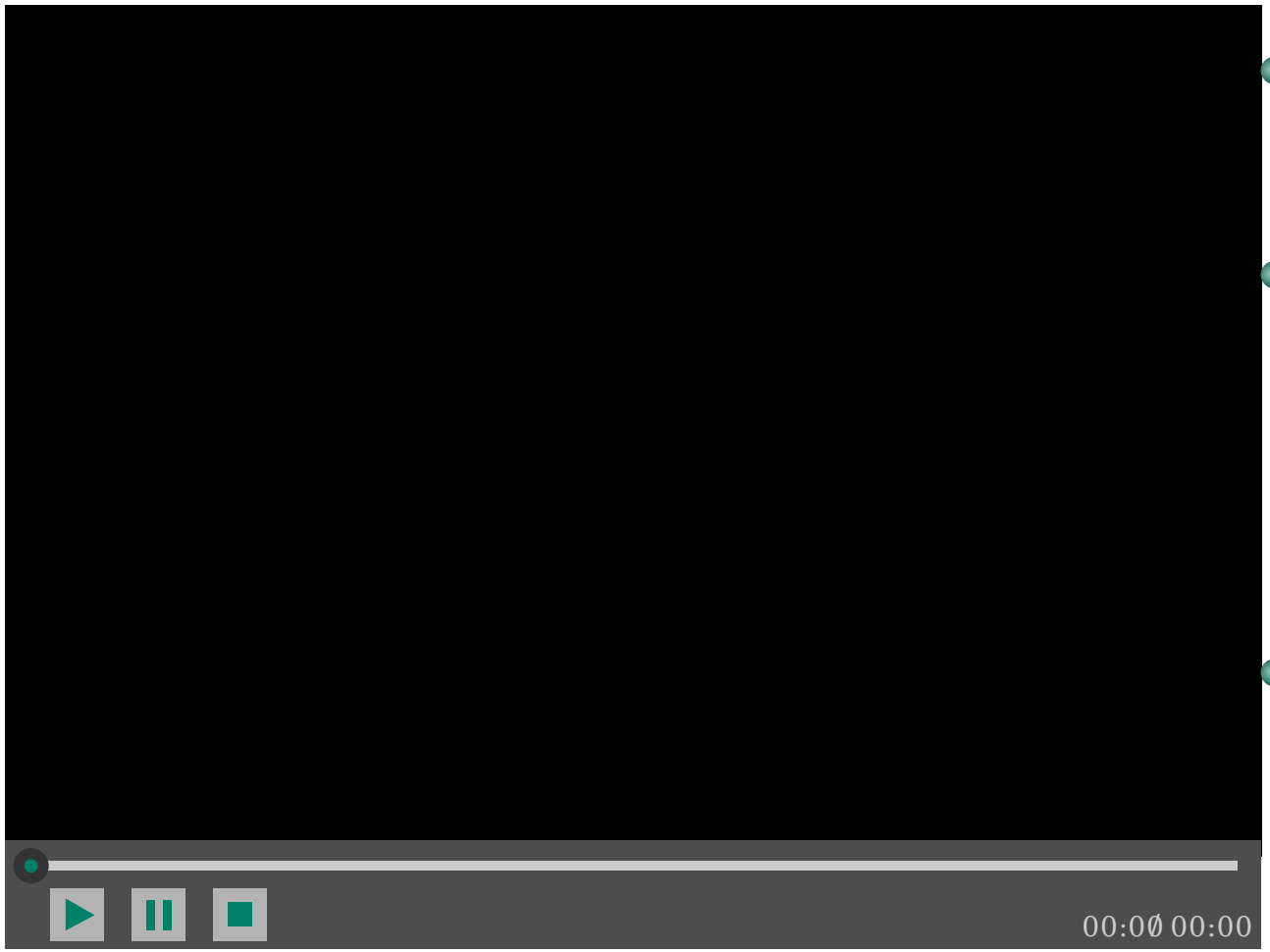
Inserindo Registros no DNS

- Exemplo: nova empresa chamada “Network Utopia”.
- Registro do domínio networkutopia.com com a entidade de registro de nomes.
  - *e.g.*, Network Solutions.
  - Necessário prover nomes e IPs dos servidores de nome autoritativos do novo domínio (primário e secundário).
  - Entidade de registro insere dois RRs na base do servidor de DNS TLD `.com`:
    - `(networkutopia.com, dns1.networkutopia.com, NS)`.
    - `(dns1.networkutopia.com, 212.212.212.1, A)`.
- No DNS autoritativo, são criadas RRs do tipo A para `www.networkutopia.com` e do tipo MX para o domínio.

Registro de um Domínio

- Usamos a ferramenta `dig` para fazer consultas ao DNS.
- Consulta inicial do tipo NS ao domínio `uff.br`.
  - Resultado: três entradas listando servidores de DNS autoritativos.
- Segunda consulta: entrada do tipo A para `server.uff.br`.
  - Resultado: endereço IP do servidor.
- Note que respostas **não são autoritativas**.
  - Possivelmente um cache do servidor DNS local utilizado.

Respostas Autoritativas *vs.* Não-autoritativas



- Mesma consulta repetida duas vezes:
  - Do tipo A para nome `www.midiacom.uff.br`.
- Inicialmente, usamos um servidor local qualquer.
  - Resultado: resposta **não-autoritativa**.
  - Possivelmente cache (pode estar desatualizada!).
- Segunda tentativa: usamos um dos servidores de DNS de `uff.br` como "servidor local"
  - Resultado: **resposta autoritativa**.

DNS e E-mail

- Como servidor de e-mail do remetente sabe qual o servidor de e-mail do destinatário?
  - Endereço de e-mail associado a um domínio.
  - *e.g.* `user@exemplo.com`.
  - Servidor do remetente faz consulta do tipo MX a domínio do destinatário.
- Consulta do tipo MX retorna um **nome**
  - Ainda precisa de uma nova resolução.
  - Consulta do tipo A.

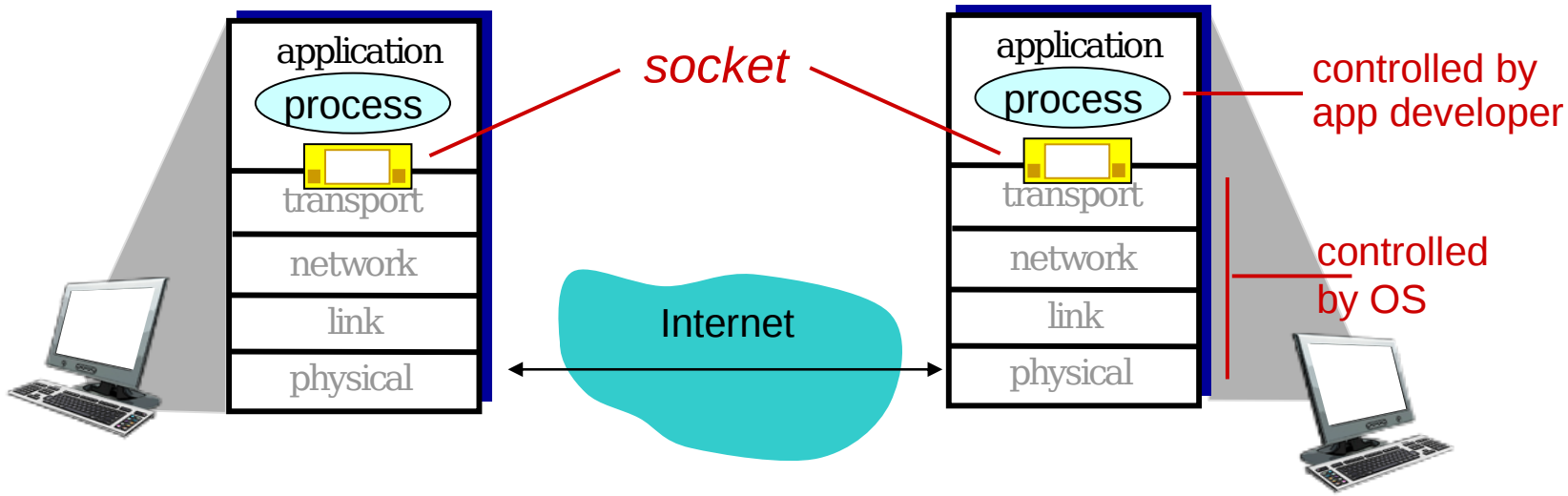
Atacando o DNS

- **Ataques de DDoS:**
  - Bombardear servidores raiz com tráfego.
    - Até hoje, não foi bem sucedido.
    - Técnicas de filtro de tráfego.
    - Servidores de DNS locais fazem cache dos IPs dos servidores TLD, evitam acessos ao servidores raiz.
  - Bombardear servidores TLD.
    - Potencialmente mais perigoso.
- **Ataques de redirecionamento:**
  - Homem-no-meio.
    - Intercepta requisições.
  - Envenenamento do DNS.
    - Envia respostas adulteradas para servidor de DNS, que faz cache das informações.
- **Exploração do DNS para DDoS:**
  - Envia requisições com IP de origem forjado (IP da vítima).
  - Requer amplificação.

Conceitos Básicos

Programação com Sockets (I)

- **Objetivo:** aprender a construir aplicações Cliente–Servidor que se comuniquem utilizando sockets.
- **Socket:** janela entre processo da aplicação e protocolo de transporte.



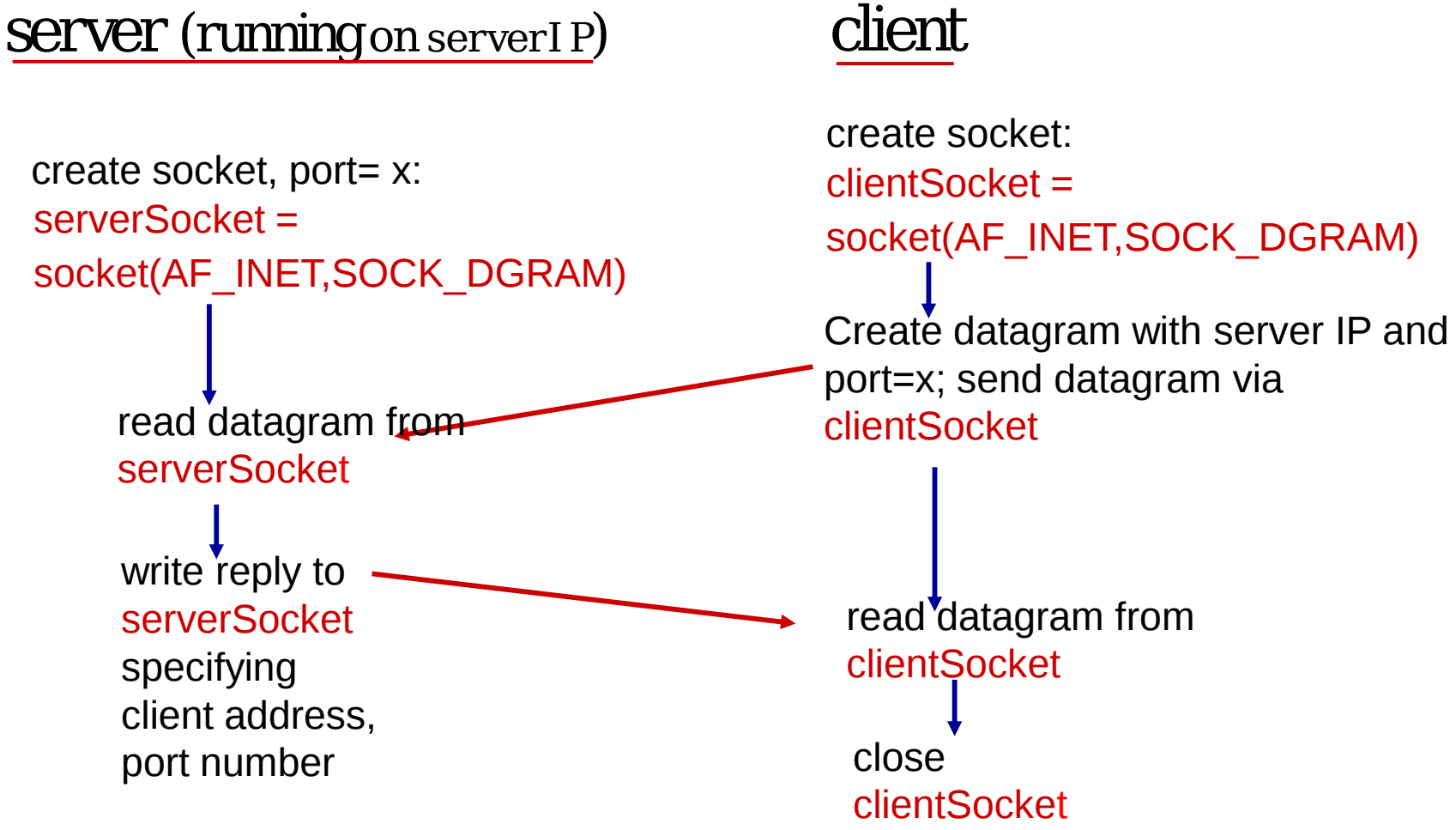
Programação com Sockets (II)

- **Dois tipos de socket para dois modelos de serviço de transporte:**
  - **UDP:** serviço de datagramas não-confiável.
  - **TCP:** serviço de entrega confiável, orientado a fluxo de bytes.
- **Aplicação de exemplo**:
  1. Cliente lê *string* do teclado e envia o dado para o servidor.
  2. O servidor recebe o dado e converte a *string* para caixa alta.
  3. Servidor envia dados modificados para o cliente.
  4. Cliente recebe dado modificado e imprime na tela.

Programação com Sockets UDP

- **UDP: não há "conexão" entre cliente e servidor.**
  - Não existe handshaking antes do envio de dados.
  - Transmissor explicitamente informa o endereço IP e o número de porta de destino a cada pacote.
  - Receptor extrai endereço IP do transmissor e número de porta do pacote recebido.
- **UDP: dados transmitidos podem ser perdidos ou recebidos fora de ordem!**
- **Ponto de vista da aplicação:**
  - UDP provê serviço **não-confiável** de transmissão de grupos de bytes ("datagramas") entre cliente e servidor.

Interação entre Cliente/Servidor e o Socket: UDP



Aplicação de Exemplo: Cliente UDP (I)

```
import java.io.*;
import java.net.*;    // API de sockets.

class UDPClient {

  public static void main(String args[]) throws Exception {

    BufferedReader inFromUser =       new BufferedReader(new InputStreamReader(System.in));
    // Criação de Socket UDP (datagramas)
    DatagramSocket clientSocket = new DatagramSocket();
    // Resolução de nome de host.
    InetAddress IPAddress = InetAddress.getByName("hostname");

    // Alocação de buffers para mensagens transmitida e recebida
    byte[] sendData = new byte[1024];
    byte[] receiveData = new byte[1024];

    // Leitura de dados do usuário
    String sentence = inFromUser.readLine();
    // Formatação da mensagem da aplicação
    sendData = sentence.getBytes();
```

Aplicação de Exemplo: Cliente UDP (II)

```
        // Criação do datagrama e envio. Note a especificação do endereço
        // de destino (IP e porta).
        DatagramPacket sendPacket =       new DatagramPacket(sendData, sendData.length, IPAddress,
        clientSocket.send(sendPacket);

        // Espera pela resposta. Funções/métodos de recepção são (normalmente) bloqueantes.
        DatagramPacket receivePacket = new DatagramPacket(receiveData, receiveData.length);
        clientSocket.receive(receivePacket);

        // Apresentação do resultado.
        String modifiedSentence = new String(receivePacket.getData());
        System.out.println("FROM SERVER:" + modifiedSentence);

        // Fechamento do socket.
        clientSocket.close();
    }
}
```

Aplicação de Exemplo: Servidor UDP (I)

```java
import java.io.*;
import java.net.*;

class UDPServer {

  public static void main(String args[]) throws Exception {

    // Criação do socket. Note que especificamos o # de porta na qual esperamos por datagramas.
    DatagramSocket serverSocket = new DatagramSocket(9876);
    byte[] receiveData = new byte[1024]; // Buffer de recepção de dados.
    byte[] sendData = new byte[1024]; // Buffer para envio de dados.

    // Servidores normalmente executam um loop infinito. Cada iteração representa o atendimento
    // a um cliente diferente.
    while(true) {

      // Criação de um datagrama para recepção de mensagem.
      DatagramPacket receivePacket = new DatagramPacket(receiveData, receiveData.length);

      // Aguardar recepção de um novo datagrama. Novamente, métodos/funções de recepção são,
      // em geral, bloqueantes.
      serverSocket.receive(receivePacket);
```

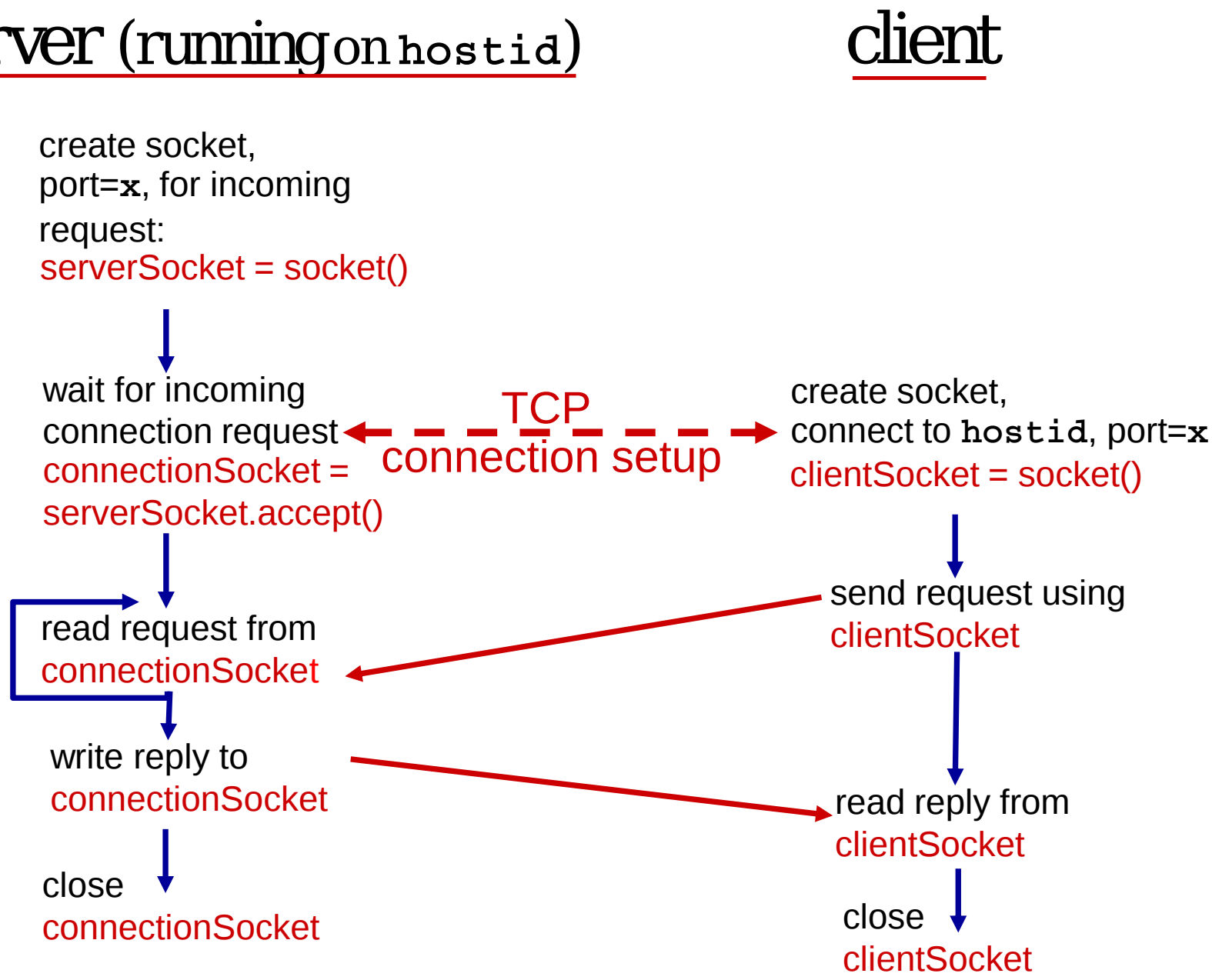
Aplicação de Exemplo: Servidor UDP (II)

```
        // Tratamento da mensagem. Aqui, é aplicada a lógica específica da aplicação.
        // No caso, apenas interpretamos os bytes da mensagem como uma string e calculamos
        // uma versão alternativa em caixa alta.
        String sentence = new String(receivePacket.getData());
        String capitalizedSentence = sentence.toUpperCase();

        // Preparação da resposta: é preciso descobrir o endereço do cliente (IP e porta).
        // Ambas as informações constam no datagrama recebido.
        InetAddress IPAddress = receivePacket.getAddress();
        int port = receivePacket.getPort();

        // Criação do datagrama de resposta. Transferimos a string para o buffer de envio e
        // construímos um datagrama a partir dele. Note, novamente, a especificação do
        // endereço de destino.
        sendData = capitalizedSentence.getBytes();
        DatagramPacket sendPacket = new DatagramPacket(sendData, sendData.length,
                                                       IPAddress, port);

        // Envio em si do datagrama.
        serverSocket.send(sendPacket);
      }
    }
  }
```

Programação com Sockets TCP

- **Cliente deve contactar servidor**.
  - Processo do servidor precisa estar previamente em execução.
  - Servidor precisa ter criado socket que aceitará contato do cliente.
- **Cliente contacta servidor:**
  - Criando socket TCP, especificando IP e número de porta do processo servidor.
  - **Quando cliente cria o socket**: TCP do cliente estabelece conexão para o TCP do servidor.
- Quando contactado pelo cliente, **TCP do servidor cria um novo socket**.
  - Novo socket utilizado para a comunicação do processo servidor com o processo cliente.
  - Este esquema de dois sockets permite ao servidor falar com múltiplos clientes.
  - Número de porta **de origem** são usados para distinguir clientes.
    - Mais detalhes no próximo capítulo
- **Ponto de vista da aplicação:**
  - TCP provê transferência confiável e ordenada de fluxo de bytes entre cliente e servidor.

Interação entre Cliente/Servidor e Socket TCP



Aplicação de Exemplo: Cliente TCP (I)

```
import java.io.*;
import java.net.*;

class TCPClient {

  public static void main(String argv[]) throws Exception {

     String sentence;
    String modifiedSentence;

    BufferedReader inFromUser = new BufferedReader(new InputStreamReader(System.in));

    // Criação do socket TCP. Note que aqui, diferentemente da versão UDP, já especificamos
    // o endereço do servidor (nome do host/ip e porta).
    Socket clientSocket = new Socket("hostname", 6789);

    // Do ponto de vista do programador, um socket TCP pode ser manipulado de forma similar
    // a um arquivo, com escrita e leitura de um fluxo de bytes.
    DataOutputStream outToServer = new DataOutputStream(clientSocket.getOutputStream());
    BufferedReader inFromServer =  new BufferedReader(new InputStreamReader(
                                                    clientSocket.getInputStream()));
    // Leitura da entrada do usuário.
    sentence = inFromUser.readLine();
```

Aplicação de Exemplo: Cliente TCP (II)

```
        // String é simplesmente "escrita" no socket. Notem que adicionamos uma quebra de linha
        // ao final da string (caractere '\n'). Isso demarcará ao servidor onde termina a mensagem
        // a ser processada.
        outToServer.writeBytes(sentence + '\n');

        // Aguardamos uma resposta do servidor. Note mais uma vez a manipulação do socket como
        // se fosse um arquivo. Aqui também uma quebra de linha denota fim da mensagem. Por fim,
        // assim como no cliente UDP, leituras são (geralmente) bloqueantes.
        modifiedSentence = inFromServer.readLine();

        // Impressão do resultado da tela.
        System.out.println("FROM SERVER: " + modifiedSentence);

        // Fechamento do socket.
        clientSocket.close();
    }
}
```

Aplicação de Exemplo: Servidor TCP (I)

```
import java.io.*;
import java.net.*;

class TCPServer {

  public static void main(String argv[]) throws Exception {
    String clientSentence;
    String capitalizedSentence;

    // Criação do socket do servidor. Este socket será usado para esperar por novas conexões.
    // Repare que especificamos um # de porta na qual desejamos esperar pelas conexões.
    ServerSocket welcomeSocket = new ServerSocket(6789);

    // Assim como o servidor UDP, servidor TCP também executa um loop infinito permitindo
    // o atendimento de múltiplos clientes.
    while(true) {

      // Função/método accept(): executada sobre socket, diz ao SO para aguardar (e aceitar)
      // novas conexões. Só faz sentindo para sockets orientados a conexão (TCP). Note que
      // o resultado da função/método é um novo socket.
      Socket connectionSocket = welcomeSocket.accept();
```

Aplicação de Exemplo: Servidor TCP (II)

```
        // O socket original é serve apenas para aguardar por novas conexões. Já o socket
        // retornado pela função/método accept representa uma conexão, realmente. É dele que
        // "leremos" os dados enviados pelo cliente e escreveremos os dados de resposta. Mais
        //  uma vez, note a abstração de arquivo.
        BufferedReader inFromClient = new BufferedReader(new
                                          InputStreamReader(connectionSocket.getInputStream()));
        DataOutputStream  outToClient = new DataOutputStream(connectionSocket.getOutputStream());

        // Aguardamos dados do cliente. Por convenção, dados terminam em uma quebra de linha.
        clientSentence = inFromClient.readLine();

        // Implementação da lógica da aplicação.
        capitalizedSentence = clientSentence.toUpperCase() + '\n';

        // Escrita do resultado no socket.
        outToClient.writeBytes(capitalizedSentence);
      }
    }
}
```

Exemplos Mais Complexos

Um (Protótipo de) Servidor Web (I)

```
import java.io.*;
import java.net.*;
import java.util.*;

class WebServer {

  public static void main(String argv[]) throws Exception  {

    // Servidor web ouve na porta 8001, ao invés da tradicional 80.
    ServerSocket listenSocket = new ServerSocket(8001);

    while(true) {
      Socket connectionSocket = listenSocket.accept(); // Aguarda conexão.

      // Tratamento da requisição está encapsulado em outro método.
      trataRequisicao(connectionSocket);
    }
  }

  private static void trataRequisicao(Socket connectionSocket) throws Exception {

    String requestMessageLine;
    String fileName;
```

Um (Protótipo de) Servidor Web (II)

```
// Criamos streams a partir do socket de conexão.
BufferedReader inFromClient =
          new BufferedReader(new InputStreamReader(connectionSocket.getInputStream()));
DataOutputStream outToClient = new DataOutputStream(connectionSocket.getOutputStream());

// Primeira linha deve informar a requisição. Ignoraremos as demais (e.g., cabeçalhos).
requestMessageLine = inFromClient.readLine();

// Campos são divididos por espaços em branco. Primeiro campo deve ser tipo
// do método. Neste protótipo, tratamos apenas requisições do tipo GET.
StringTokenizer tokenizedLine = new StringTokenizer(requestMessageLine);
if (tokenizedLine.nextToken().equals("GET")){

    // Próximo campo é o caminho do objeto requisitado.
    fileName = tokenizedLine.nextToken();

    // Arquivos servidos pelo servidor web são confinados ao diretório do qual ele é
    // executado (e subdiretórios). Logo, se a requisição referencia um caminho
    // absoluto (i.e., iniciado por '/'), precisamos transformar isso em um caminho
    // relativo ao diretório corrente.
    if (fileName.startsWith("/") == true)
      fileName  = fileName.substring(1);
```

Um (Protótipo de) Servidor Web (III)

```
File file = new File(fileName);
if (file.exists()) {
  int numOfBytes = (int) file.length();
  FileInputStream inFile  = new FileInputStream (fileName);
  byte[] fileInBytes = new byte[numOfBytes];
  inFile.read(fileInBytes);

  // Composição da mensagem de resposta: começamos com a linha de status.
  outToClient.writeBytes("HTTP/1.0 200 OK\r\n");
  // Precisamos de algumas linhas de cabeçalho na resposta. A primeira para
  // informar o tipo do arquivo.
  if (fileName.endsWith(".jpg")) outToClient.writeBytes("Content-Type: image/jpeg\r\n");
  else if (fileName.endsWith(".gif")) outToClient.writeBytes("Content-Type: image/gif\r\n");
  else if (fileName.endsWith(".html")) outToClient.writeBytes("Content-Type: text/html\r\n");
  // ...
  // Outra linha de cabeçalho: tamanho do conteúdo anexado ao corpo da resposta.
  outToClient.writeBytes("Content-Length: " + numOfBytes + "\r\n");
  // Cabeçalhos são separados do corpo por uma linha em branco no HTTP.
  outToClient.writeBytes("\r\n");
  // Colocamos os bytes do objeto no corpo da mensagem.
  outToClient.write(fileInBytes, 0, numOfBytes);
}
```

Um (Protótipo de) Servidor Web (IV)

```
            else {
              // Objeto não encontrado.
              outToClient.writeBytes("HTTP/1.0 404 Not Found\r\n");
            }
          }
          // Tratamento (muito básico) de erros.
          else System.out.println("Bad Request Message");

          // Fechamos o socket da conexão.
          connectionSocket.close();
        }
      }
```

Ferramenta de Medição de Vazão TCP: Cliente (I)

```java
import java.io.*;
import java.net.*;

class BWTestClient {

  public static void main(String argv[]) throws Exception {

    byte buffer[] = new byte[8192];
    int i = 0;
    long endTime, now;

    // Criação do socket TCP. Note que aqui, diferentemente da versão UDP, já especificamos
    // o endereço do servidor (nome do host/ip e porta).
    Socket clientSocket = new Socket("localhost", 6789);

    // Do ponto de vista do programador, um socket TCP pode ser manipulado de forma similar
    // a um arquivo, com escrita e leitura de um fluxo de bytes.
    DataOutputStream outToServer = new DataOutputStream(clientSocket.getOutputStream());
    BufferedReader inFromServer =  new BufferedReader(new InputStreamReader(
                                                    clientSocket.getInputStream()));
```

Ferramenta de Medição de Vazão TCP: Cliente (II)

```
    // Armazenar hora do final do teste (testes sempre têm 10 segundos).
    endTime = System.currentTimeMillis() + 10000;

    // Simplesmente escrevemos continuamente no socket. Escritas *normalmente* não são
    // bloqueantes, mas o TCP limitará a taxa de envio de acordo com a capacidade da
    // rede. Quando excedermos esta capacidade, a chamada bloqueará.
    while(true) {
      outToServer.write(buffer);
      i = i + 1;
      now = System.currentTimeMillis();
      if (now >= endTime) break ;
    }

    // Impressão do resultado da tela. A cada iteração do loop anterior, transmitimos
    // 64 kb. Para calcular vazão, basta multiplicar i por 64 e dividir por 10.
    System.out.println("Vazão (kb/s): " + (i * 64 / 10.0));

    // Fechamento do socket.
    clientSocket.close();
  }
}
```

Ferramenta de Medição de Vazão TCP: Servidor (I)

```
import java.io.*;
import java.net.*;

class BWTestServer {

  public static void main(String argv[]) throws Exception {

    char buffer[] = new char[8192];

    // Criação do socket do servidor. Este socket será usado para esperar por novas conexões.
    // Repare que especificamos um # de porta na qual desejamos esperar pelas conexões.
    ServerSocket welcomeSocket = new ServerSocket(6789);

    // Assim como o servidor UDP, servidor TCP também executa um loop infinito permitindo
    // o atendimento de múltiplos clientes.
    while(true) {

      // Função/método accept(): executada sobre socket, diz ao SO para aguardar (e aceitar)
      // novas conexões. Só faz sentindo para sockets orientados a conexão (TCP). Note que
      // o resultado da função/método é um novo socket.
      Socket connectionSocket = welcomeSocket.accept();
```

Ferramenta de Medição de Vazão TCP: Servidor (II)

```
        // O socket original é serve apenas para aguardar por novas conexões. Já o socket
        // retornado pela função/método accept representa uma conexão, realmente. É dele que
        // "leremos" os dados enviados pelo cliente e escreveremos os dados de resposta. Mais
        //  uma vez, note a abstração de arquivo.
        BufferedReader inFromClient = new BufferedReader(new
        InputStreamReader(connectionSocket.getInputStream()));

        // Simplesmente, aguardamos dados do cliente, indefinidamente.
        while(true) {

          try {

            if (inFromClient.read(buffer) < 0) break ;
          }
          catch(IOException e) {

            // Cliente fechou a conexão.
            break ;
          }
        }
      }
    }
  }
```

Outras Linguagens: Funções/Métodos Típicos

Funções/Métodos Tipicamente Utilizados

Cliente

- **socket():** criar novo socket de um determinado tipo.
- **write():** "passa" dados/mensagens pelo socket p/ transporte.
- **sendto():** envia mensagem por socket sem conexão (UDP).
- **read():** "recebe" dados/mensagens pelo socket do transporte.
- **recvfrom():** recebe mensagem por socket sem conexão (UDP).
- **connect():** abre uma conexão (TCP) para servidor/porta especificados.
- **getByName() ou getHostByName():** resolve nome para endereço IP.
- **close():** fecha o socket (e conexão, se aplicável).

Servidor

- **socket():** criar novo socket de um determinado tipo.
- **write():** "passa" dados/mensagens pelo socket p/ transporte.
- **sendto():** envia mensagem por socket sem conexão (UDP).
- **read():** "recebe" dados/mensagens pelo socket do transporte.
- **recvfrom():** recebe mensagem por socket sem conexão (UDP).
- **bind():** associa socket à porta especificada.
- **listen():** habilita socket (TCP) a receber conexões.
- **close():** fecha o socket (e conexão, se aplicável).

Sockets em Outras Linguagens: Python (Cliente TCP)

```
from socket import *

serverName = 'servername'
serverPort = 12000

# Criação do socket
clientSocket = socket(AF_INET, SOCK_STREAM)
# Conexão com o servidor
clientSocket.connect((serverName,serverPort))

sentence = raw_input('Input lowercase sentence:')
# Envio de bytes
clientSocket.send(sentence)

# Recepção
modifiedSentence = clientSocket.recv(1024)
print 'From Server:', modifiedSentence

# Fechamento
clientSocket.close()
```

Sockets em Outras Linguagens: Python (Servidor TCP)

```
from socket import *

serverPort = 12000

# Criação do socket, associação à porta 12000 e habilitar escuta por conexões
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind((‘’,serverPort))
serverSocket.listen(1)

print ‘The server is ready to receive’

while 1:

  # Aguardar nova conexão
  connectionSocket, addr = serverSocket.accept()
  # Recepção de dados
  sentence = connectionSocket.recv(1024)
  capitalizedSentence = sentence.upper()
  # Envio
  connectionSocket.send(capitalizedSentence)
  # Fechamento
  connectionSocket.close()
```