

CAN201 - Week 7

Network Layer - Addressing, NAT/IPv6, SDN & Routing Algorithms

InubashiriLix (Github)

0. 大图：网络层这一讲在整个课程中的位置

0.1 分层回顾 & 本讲定位

- 课程整体自顶向下：
 1. Application Layer
 2. Transport Layer (TCP/UDP)
 3. Network Layer ← **Week 6-7 重点**
 4. Link Layer
 5. Physical Layer
- 网络层有两条主线：
 1. * 数据平面 *：单个路由器怎样转发一个数据报 (forwarding)
 2. * 控制平面 *：全网怎样决定应该走哪条路 (routing)
- Week 7 主要解决四个问题：
 1. *“谁是谁”：IP addressing & subnet* (一个接口一个 IP)
 2. “怎么接入”：DHCP & 地址分配机制
 3. “地址不够了怎么办”：NAT & IPv6 & tunnelling
 4. “如何选路”：routing algorithms (Link State vs Distance Vector)

1. 网络层服务 & 两个核心动作

1.1 网络层提供什么服务？

- * 目标 *：在“主机之间”传递 * 数据报 (datagram)*，服务上传输层 (TCP/UDP)。
- 端系统里：
 - 发送方：把 TCP/UDP segment 封装进 IP datagram，交给链路层；
 - 接收方：从 IP datagram 中解出 segment，交给 TCP/UDP。
- 所有网络设备上都有网络层：
 - host：只对本机收发；
 - router：只看 IP 头/路由表，完成转发。

1.2 两个关键词：forwarding vs routing

- *Forwarding* (转发) — “在一个十字路口怎么走”：
 - 输入：一个到达特定路由器输入端口的数据报；
 - 依据：数据报头部 (主要是目的 IP) + 转发表 (forwarding table)；
 - 输出：送往哪个输出端口。
- *Routing* (路由) — “从 A 到 B 的整条旅游路线怎么规划”：
 - 输入：网络拓扑 + 链路代价；
 - 输出：每对源-目的之间的“好路径” (通常最小代价路径)；
 - 由 *routing protocol* (如 OSPF、BGP) 或 **SDN controller** 生成转发表。
- 心智模型：
 - * 数据平面 *：局部，按包做 forwarding；
 - * 控制平面 *：全局，算 routing，改 forwarding table。

2. IP 地址、接口与子网：谁是谁、谁跟谁同一网

2.1 “接口 (interface)” 才有 IP 地址

- * 接口 *：路由器/主机与物理链路的连接点：
 - 路由器：通常有多个接口 (每个口连一个子网)；
 - 主机：常有 1-2 个接口 (有线 + WiFi)。
- *IP 地址 *：32-bit 标识，每个 接口一个地址 (不是“每台设备一个地址”)：
 - 例如：223.1.1.1 = 11011111 00000001 00000001 00000001；
 - 对人类用点分十进制：223.1.1.1。
- Q：路由器有 5 个接口，有几个 IP 地址？
 - A：5 个，每个接口一个。

2.2 子网 (subnet)：物理上连在一起的一坨

- * 子网定义 *：
 - 一组接口，它们之间可以“直接在二层互通”，不需要经过路由器。
 - 从图上看：把所有路由器接口“拔掉”，剩下的一块块“岛”，每块就是一个子网。
- IP 地址划分为两部分：
 1. * 网络 / 子网部分 * (高位)；
 2. * 主机部分 * (低位)。
- 记法：CIDR 前缀 ‘a.b.c.d/x’
 - /x 表示前 x 比特是网络部分；
 - 示例：223.1.1.0/24 表示：
 - 网络前缀：223.1.1；
 - 主机部分：最后 8 bit；
 - 地址范围：223.1.1.0-223.1.1.255。

2.3 手工找子网的“recipe” (心智步骤)

1. 把路由器接口从连线中“剪掉”——接口本身不属于任何子网；
 2. 留下的是一块块二层连通的“岛”——每块是一个子网；
 3. 对每块子网：
 - 取该子网内所有接口 IP 地址的公共高位部分作为网络前缀；
 - 把网络前缀 + “/前缀长度” 记为子网地址，比如 223.1.3.0/24。
- 扩展练习：
 - 给定一张图，按这种方式数出有几个 /24 子网，各自的前缀是什么。

2.4 从 Classful 到 CIDR：为什么要“classless”

- 早期 *classful addressing*：
 - A 类：/8、B 类：/16、C 类：/24；
 - 问题：
 - 一个 B 类 /16 支持 65,534 主机，对一个只需要 10,000 主机的组织来说太浪费；
 - 大量 B 类被申请却浪费，导致地址空间迅速耗尽。

- *CIDR (Classless Inter-Domain Routing)*:
 - 打破 A/B/C 边界，网络部分长度可以是任意 x；
 - 提高地址空间利用率，也利于路由聚合（后面会讲）。

3. 地址是怎么“发下去”和“动态分给主机”的？

3.1 第 1 个问题：主机的 host 部分怎么来？—— DHCP

- 静态方式：
 - 管理员手动配置：IP、子网掩码、默认网关、DNS 等；
 - 小网络可以，大网络和移动设备场景不现实。
- DHCP (Dynamic Host Configuration Protocol):
 - * 目标 *：主机接入网络时自动获得：
 - IP 地址 (host 部分)；
 - 默认网关 (first-hop router)；
 - DNS 服务器地址；
 - 子网掩码；
 - 以及租期 (lease time) 等信息。

3.2 DHCP 四步握手（广播，广播，广播，广播）

- 典型场景（新主机插上线）：
 1. *DHCPDISCOVER*:
 - src IP: 0.0.0.0, src port 68;
 - dst IP: 255.255.255.255, dst port 67;
 - 主机广播：“网里有 DHCP 服务器吗？”
 2. *DHCPOFFER*:
 - DHCP 服务器广播回应，提供一个可用 IP + 租期；
 3. *DHCPREQUEST*:
 - 主机广播请求：“好的，我想用你刚才给我的这个 IP”；
 4. *DHCPACK*:
 - 服务器广播确认，该 IP 正式分配给这台主机一段时间（租期）。
- 为什么要广播？
 - 新主机一开始还没有合法 IP 和网关，只能靠广播把包送进局域网；
 - 服务器要用广播回应，确保“还没完全配好的主机”也能收到。

3.3 第 2 个问题：一个“网络”的网络前缀怎么来？—— ISP & ICANN 分配

- 组织网络（比如一个大学）不会直接从 ICANN 拿地址，而是：
 - 上游 ISP 拿到一个比较大的块，比如 200.23.16.0/20；
 - 再把它拆成若干 /23 或 /24 分配给多个组织：
 - Org0: 200.23.16.0/23
 - Org1: 200.23.18.0/23
 - ...
- ICANN 负责：
 - 把全球 IPv4 空间分给 5 个区域注册局 (RIR)；
 - 管理 DNS 根、TLD 等。

4. 分级 (hierarchical) 地址与路由聚合

4.1 路由聚合：用一个前缀代表一大片

- ISP 拿到的块：200.23.16.0/20，底下有多个 /23 组织。

- 向网络中其它 AS 通告时：
 - ISP 只需要广告一条“200.23.16.0/20 都从我这里来”；
 - 而不是单独广告每个 /23。
- 结果：
 - 路由表条目更少；
 - 更易扩展。

4.2 “更具体的路由”覆盖“宽泛路由”

- 假设 Org1 从 ISP-A 迁移到 ISP-B:
 - 原来 ISP-A 广告 200.23.16.0/20；
 - 现在 ISP-B 广告一个更具体的前缀：200.23.18.0/23。
- 路由选择规则：* 最长前缀匹配 * (Longest Prefix Match):
 - 对于目的地址 200.23.18.x:
 - 同时匹配 /20 和 /23；
 - 选 /23 (更具体)。
- 效果：
 - 路由聚合仍然存在 (/20)，个别迁移的子块再用更具体前缀覆盖。

5. NAT: IPv4 地址不够时的“翻译器”

5.1 NAT 的核心思路：局域网内部“私有地址”，外面只看见一个公网 IP

- 私有地址空间（仅在局域网使用）：
 - 10.0.0.0/8
 - 172.16.0.0/12
 - 192.168.0.0/16
- NAT 场景：
 - 家用路由上联 ISP 给的一个公网 IP: 138.76.29.7；
 - 内网设备用 10.0.0.x 私有地址；
 - 对外所有连接都看起来来自 138.76.29.7（配合不同端口号）。

5.2 NAT 的 translation table 思维模型

- 对外发包时：
 1. 内网主机发出: src=(10.0.0.1, 3345), dst=(128.119.40.186, 80);
 2. NAT 路由器：
 - 分配一个新的外部端口，如 5001；
 - 把源改写为 (138.76.29.7, 5001)；
 - 在 NAT 表记一条映射：
 - (WAN: 138.76.29.7, 5001) ↔ (LAN: 10.0.0.1, 3345)；
 3. 外部服务器回复: dst=(138.76.29.7, 5001)；
 4. NAT 查表，改回 dst=(10.0.0.1, 3345) 转发给内网主机。

5.3 NAT 的优缺点心智模型

- 优点：
 - 节省公网地址：一个公网 IP 容纳成百上千内网主机；
 - 改内网地址不影响外部；
 - 换 ISP 只改 NAT 外部 IP，内网不动；
 - 一定程度上 * 隐藏内网结构 *，增加安全性。
- 争议点：
 - 路由器（网络层设备）修改了 * 传输层端口号 *，打破“端到端原则”；
 - 给 P2P / VoIP / 服务器在 NAT 后面带来“打洞”的复杂性；
 - 理论上 IPv6 才是根本解决方案，但现实中 NAT 已经无处不在（家庭、企业、4G/5G）。

6. IPv6 与 Tunnelling: 地址耗尽 + 升级路径

6.1 IPv6 的动机与变化

- IPv4 问题:
 - 32-bit 地址空间不足 (已在 2011 年分配完所有大块);
 - header 有 checksum、options、fragmentation 等影响性能。
- IPv6 关键变化:
 - 地址变为 128 bit (几乎无穷);
 - fixed-length 40-byte 头部:
 - 没有 IP 头 checksum;
 - 没有中间路由器 fragmentation (路由器不分片);
 - 扩展功能通过 扩展头部 / 上层协议实现。

6.2 IPv6 数据报结构 (与 IPv4 对比记忆)

- IPv6 header 大致包含:
 - version, traffic class/priority, flow label;
 - payload length;
 - next header (指 TCP/UDP/扩展头);
 - hop limit (类似 IPv4 的 TTL);
 - 128-bit source / destination address。
- “消失”的东西 (和 IPv4 对比记忆):
 - 不再有 IP 头部 checksum;
 - 不再有 fragmentation fields;
 - 不再有 “options” 字段 (改为 extension headers)。

6.3 升级难点 & Tunnelling 解决方案

- 现实世界无法 “某一天所有路由器一起升级 IPv6”;
- 所以会长时间存在 “IPv4 + IPv6 混合网络”:
 - 有的路由器只懂 IPv4;
 - 有的支持双栈 IPv4/IPv6。
- Tunnelling (隧道) 心智模型:
 - 把 IPv6 数据报当成 “payload”, 封装进 IPv4 数据报;
 - 在 IPv4-only 网络中 “穿过”;
 - 隧道两端是 IPv6 路由器, 负责封装和解封。
- 逻辑视图:
 - A → B (IPv6) → IPv4-only 区域 → E → F (IPv6);
 - B 在入隧道处: 把 IPv6 datagram 封在 IPv4 datagram 里;
 - E 在出隧道处: 解封, 还原 IPv6 datagram 继续正常转发。

7. Generalized Forwarding & SDN/OpenFlow: match+action 心智模型

7.1 Flow & Flow Table 抽象

- *Flow*: 由报文头字段组合定义的一类流量 (不限于 IP):
 - 可以用 link 层 (MAC)、network 层 (IP)、transport 层 (port)、甚至 VLAN ID 等定义。
- generalized forwarding
 - 把路由、交换、防火墙、NAT 统一抽象为:
 - *match*: 匹配报文头字段模式 (pattern);
 - *action*: 对匹配的报文执行操作:
 - forward 到某端口;
 - drop;
 - 修改头字段 (如 NAT);
 - 送给控制器处理;
 - *priority*: 多条规则重叠时谁优先;

- *counters*: 统计匹配字节数、报文数。

7.2 OpenFlow 规则表的 “字段视图”

- 一条 OpenFlow 表项大致包括:
 - 匹配字段 (可能包含通配符 *):
 - 端口号、MAC 源/目的、Eth type、VLAN ID;
 - IP 源/目的、IP 协议号;
 - TCP/UDP 源/目的端口;
 - 以及 QoS 字段 (ToS/DSCP 等)。
 - 动作:
 - forward(port X);
 - drop;
 - send to controller;
 - 修改字段 (如改 IP/port 实现 NAT);
 - 计数器: 匹配的包数/字节数。
- 例子:
 - * 目的 IP 为 51.6.0.8 的 都从 6 号口发出 *:
 - 匹配: IPdst = 51.6.0.8, 其余字段 *;
 - 动作: forward(6)。
 - * 阻止所有去 TCP 端口 22 (SSH) 的流量 *:
 - 匹配: TCP dport = 22, 其余字段 *;
 - 动作: drop。

7.3 “统一抽象” 的心智模型

- 传统设备在 match+action 框架下的定位:
 - Router: match = 目的 IP 前缀, action = 转发到某端口;
 - L2 Switch: match = 目的 MAC, action = 转发/洪泛;
 - Firewall: match = IP + port + protocol, action = permit/deny;
 - NAT: match = IP+port, action = 重写 IP+port;
- 统一之后:
 - 控制平面 (如 SDN controller) 只需发规则;
 - 数据平面 (交换机/路由器) 统一执行 match+action。

8. Routing Algorithms: 图抽象 + LS vs DV

8.1 Graph 抽象: 把网络看成带权图

- 用图 $G = (N, E)$ 表示网络:
 - N: 节点集合 (routers);
 - E: 边集合 (链路), 每条边 (x, y) 有一个 链路代价 $c(x, y)$:
 - 可以简单设为 1 (hop 数);
 - 或按带宽、时延、拥塞程度设定。
- 路径 cost:
 - 路径 $x_1 \rightarrow x_2 \rightarrow \dots \rightarrow x_p$ 的代价为:
 - $c(x_1, x_2) + c(x_2, x_3) + \dots + c(x_{p-1}, x_p)$ 。
- 路由问题:
 - 给定源 u 和目的 z , 找到 * 最小代价路径 *;
 - 或对所有目的节点, 给出最小代价路径。

8.2 路由算法分类: 全局 vs 分布式

- *Link-State (LS) / 全局信息算法 *:
 - 每个路由器都掌握完整拓扑和所有链路代价;
 - 通过 link-state 广播获得这张 “全网地图”;

- 各自在本地运行同样的算法 (Dijkstra) 算出到所有目的的最短路;
- 典型协议: OSPF、IS-IS。
- *Distance-Vector (DV) / 分布式算法 *:
- 每个路由器只知道自己到所有目的“距离估计表” (distance vector);
- 周期性/触发式地把自己的 vector 发给邻居;
- 接收邻居的 vector 后用 Bellman-Ford 公式更新;
- 典型协议: RIP、BGP (思想类似, 细节更复杂)。
- 另一个维度: * 静态 * vs * 动态 *:
- 静态: 配置很少改, 路由表基本不变;
- 动态: 链路成本/拓扑变化时会更新 (现实互联网几乎都用动态)。

9. Dijkstra Link-State Algorithm: 有“地图”的最短路

9.1 输入与输出的心智模型

- 输入:
 - 完整的图 $G = (N, E)$;
 - 每条边 $c(x, y)$;
 - 源节点 u 。
- 输出:
 - u 到所有节点 v 的最小代价 $D(v)$;
 - 对每个 v 的“前驱节点” $p(v)$, 从而构成 * 最短路径树 * (shortest path tree);
 - 由此决定转发表: 对每个目的 v , 从 u 应该走哪个 * 下一跳 *。

9.2 核心思想: 一次确定一个节点的“最终最短路”

- 维护集合 N' : 已经求出“最终最短路径”的节点集合:
 - 一开始: $N' = \{u\}$ (源节点到自身的距离 0)。
- 对其余每个节点 v :
 - 维护 $D(v)$: 当前已知的从 u 到 v 的路径最小代价估计;
 - 初始: 若 v 与 u 直接相邻, 则 $D(v) = c(u, v)$, 否则 $D(v) = \infty$ 。
- 迭代步骤:
 1. 在所有不在 N' 中的节点中, 选一个 $D(w)$ 最小的 w ;
 2. 把 w 加入 N' (说明 $u \rightarrow w$ 的最短路径已确定);
 3. 用 w * 松弛 * (relaxation) 其邻居 v :
 - $D(v) \leftarrow \min\{D(v), D(w) + c(w, v)\}$;
 - 如果通过 w 的新路径更短, 就更新 $D(v)$ 和前驱 $p(v)$;
 4. 重复, 直到所有节点都被加入 N' 。
- 复杂度:
 - 朴素实现: $O(N^2)$;
 - 用堆优化可到 $O(E \log N)$ 。

9.3 如何从结果构造转发表?

- 对源 u :
 - 最短路径树告诉你: 每个目的 v 的路径 $u \rightarrow \dots \rightarrow v$;
- 对每个目的 v :
 - 找到路径上 u 后面紧接着的那个节点 w ;
 - 在转发表中记录: $\text{dest}=v$, $\text{next-hop}=w$, $\text{outgoing-interface}=if(w)$ 。

10. Distance-Vector & Bellman-Ford: 只看“邻居报价”的最短路

10.1 Bellman-Ford 方程: 递归定义最短路

- 定义 $d_x(y)$: 从节点 x 到 y 的 * 最小代价 *。
- Bellman-Ford 公式:
 - 对任意 $x \neq y$, 有:
 - $d_x(y) = \min \text{ over } v \in \text{Neighbors}(x) \{ c(x, v) + d_v(y) \}$ 。
 - 解释:
 - 从 x 去 y 的最短路, 第一跳一定是某个邻居 v ;
 - 于是代价 = 到 v 的直接代价 + 从 v 到 y 的最短代价。

10.2 Distance Vector 算法的“行为模式”

- 每个节点 x 维护一个表:
 - $D_x(y)$: 当前估计的从 x 到所有 y 的最小代价;
 - 初始:
 - $D_x(x) = 0$;
 - $D_x(y) = c(x, y)$ 若 y 为邻居;
 - 否则 $D_x(y) = \infty$ 。
- 迭代过程 (异步/分布式):
 1. “不时”地, 每个节点把自己的 $D_x(*)$ 发给所有邻居;
 2. x 收到某邻居 v 的 $D_v(*)$ 后, 对每个目的 y 做:
 - $D_x(y) \leftarrow \min\{D_x(y), c(x, v) + D_v(y)\}$;
 3. 若 $D_x(y)$ 有变化, 则 x 再把新的 $D_x(*)$ 广播给邻居。
- 在“自然条件”下:
 - 所有 $D_x(y)$ 会收敛到真正最小代价 $d_x(y)$;
 - 收敛后, 转发表可以根据“哪一个邻居 v 提供了最小代价”来确定下一跳。

10.3 小三节点示意例子 (x, y, z)

- 三个节点 x, y, z , x 与 y, z 相邻, y 与 z 相邻;
- 初始时:
 - y 知道自己到 z 的代价 1;
 - x 知道自己到 y 的代价 2、到 z 的代价 7;
 - z 知道自己到 y 的代价 1;
- 通过 DV 交换后, x 可以算出经由 y 到 z 更便宜:
 - $D_x(z) = \min\{c(x, y) + D_y(z), c(x, z) + D_z(z)\} = \min\{2+1, 7+0\} = 3$;
 - 从而更新 $x \rightarrow z$ 的下一跳为 y 。

10.4 DV 的优点与典型问题 (心智模型)

- 优点:
 - * 分布式, 易扩展 *: 每个节点只和邻居说话;
 - 不需要全局拓扑信息;
 - 算法简单。
- 问题:
 - *count-to-infinity* (数到无穷) 问题: 当某条链路失效时, “坏消息传得慢”, 各种节点互相给出错误估计导致距离不断增加;
 - 实际协议引入诸如:
 - poisoned reverse;
 - infinity 设上限 (如 RIP 用 16 表示“不可达”);
 - 触发式更新等。

11. 总结: Week 7 整体心智模型

1. ** 地址视角 **: IP addressing & subnets
 - 一个接口一个 IP, 子网 = 二层连通的一坨;
 - CIDR 使得网络前缀长度可变, 利于高效分配与路由聚合。
2. ** 主机接入视角 **: DHCP & ISP 分配
 - 主机通过 DHCP 动态获得 host 部分 + 默认网关 + DNS;
 - 组织通过 ISP 从更大的前缀块中拿到 network 部分。
3. ** 地址枯竭与演进 **: NAT & IPv6 & Tunnelling
 - NAT: 用私有地址 + 端口映射, 大量设备共享一个公网 IP;
 - IPv6: 从根本上扩展地址空间、简化头部, 加速转发;
 - Tunnelling: 在 IPv4-only 区域内 “封装” 传输 IPv6 数据报。
4. ** 数据平面抽象 **: generalized forwarding & Open-Flow
 - 一切转发设备都可以看成 “match+action” 的 flow table;
 - 路由器、交换机、防火墙、NAT 在这一框架下统一建模。
5. ** 控制平面算法 **: Routing Algorithms
 - 把网络抽象为带权图;
 - Link-State (Dijkstra): 每个节点有全局地图, 局部跑最短路;
 - Distance Vector (Bellman-Ford): 只向邻居交换 “报价”, 逐渐收敛;
 - 二者分别对应 OSPF / RIP 等实际协议。

一句话压缩 Week 7:

“先帮你搞清谁是谁 (IP/子网/DHCP/地址分配), 再想办法在地址不够时苟下去 (NAT/IPv6/tunnel), 然后用一个 match+action 的视角统一所有网络设备 (SDN/OpenFlow), 最后用图论和 Dijkstra/Bellman-Ford 解释控制平面到底是怎么把 ‘路’ 算出来的。”