Ministerul Educației al Republicii Moldova

Universitatea Tehnică a Moldovei

Facultatea CIM

Filiera Anglofonă

Raport:

Lucrarea de laborator №1

La Rețele de Calculatoare

Tema: Configurararea rețelelor Ethernet

A elaborat: *Studentul grupei FAF-071(l. engleză)*
Ivanov Sergiu

A verificat: *Romanenko Alexandr*

Chişinău 2010

Topic: Configuration of Ethernet Networks

Objective: Study the standard-driven practical aspects of the physical configuration of Ethernet-type local area networks.

1 The Basics

Designing an Ethernet based on a single segment of a given media variety is pretty straightforward. To stay within the specifications when using thick Ethernet, for example, you must make sure that the thick Ethernet cable segment does not exceed 500 meters in length and does not contain more than 100 MAU connections to the cable. As long as the cable that you use to build the segment also meets the media specifications and is properly installed, the system will work correctly.

However, when you build a multi-segment network things get more complex. To help you maintain compliance with the Ethernet specifications when combining segments from a variety of Ethernet media types, the IEEE developed two models for verifying the operation of a multi-segment Ethernet. Transmissi on System Model 1 provides a set of standard configuration rules, and Transmission System Model 2 provides a set of calculation aids so that you can do the calculations yourself.

The scope of the multi-segment configuration guidelines is limited to a single Ethernet, or "collision domain." A collision domain is formally defined as a single CSMA/CD network in which there will be a collision if two computers attached to the system both transmit at the same time. To combine multiple segments into a single Ethernet system you use repeaters, MAUs, AUI cables, and the segments themselves to create a network that functions as a single collision domain. Each computer, or DTE, contains an Ethernet interface that implements the medium access control (MAC) rules for Ethernet. If two DTEs with their associated Ethernet interfaces and MACs are attached to segments that are connected by repeaters then they are within the same collision domain, since rep eaters are designed to propagate collisions onto all segments to which they are connected.

If two DTEs are instead separated by a packet switch such as a bridge or router, then they are in separate collision domains, since packet switches do not propagate collisions. Instead, bridges and routers contain multiple Ethernet interfaces and are designed to receive a packet on one Ethernet and transmit the data onto another Ethernet in a new packet.

Instead of propagating collision signals between Ethernets, packet switches interrupt the collision domain and provide separation for the operation of the Ethernets they link. Therefore, you can use packet switches to build larger network systems by connecting individual Ethernet systems. The point is that the configuration guidelines apply to a single collision domain only, and have nothing to say about combining multiple Ethernets with packet switches.

An Ethernet network can operate in two modes:

a) *Half-duplex mode* (multipoint) – This is the usual way. This implies sharing by the hosts of a collision domain of a single transimission medium, using the CSMA/CD access method. The "multipoint" part is normally omitted.
b) *Full duplex mode* – This is a point-to-point mode of operation by which each host is connected to a bridge via two channels. Thus the transmissions towards the network is done by one channel and the transmission in the opposite direction is done by another channel. In this mode the network works stably even at relatively high load.

For an Ethernet network to function correctly it is first of all necessary that the restrictions established for the physical medium be respected. These include the maximal length of a segment, the number of repeaters between two hosts which belong to the same collision domain, the maximal diameter of the network, etc. There are several empirical rules for constructing a robust network, but specialists normally have the information from the standards at hand. This allows to specify the conditions of normal functioning of the normal more exactly. This data is especially useful in hybrid networks, which the usual situation in local area networks.

For an Ethernet network to function correctly the following four basic conditions should be satisfied:

a) the maximal number of hosts in the network should be less than 1024;
b) the physical restrictions established for every standard should be strictly respected;
c) the path delay value (PDV) between two most distant hosts should not be greater than $575bt$;
d) the path variability value (PVV) after the frame sequence has come through all repeaters should not be greater than $49bt$.

2 Practical Assignment

2.1 Network Structure

A graphical representation of the structure of the network suggested to me in this laboratory assignment is shown in Figure 1.

2.2 Computing PDV and PVV Automatically

In order to determine whether the network complies with the standards, I wrote a program which computes the PDV and PVV values for all pairs of hosts and then chooses the maximal value for each of these parameters. The maximal values obtained for the suggested network are:

a) PDV: $568.748bt$ between stations 6 and 2.
b) PVV: $46.000bt$ between stations 2 and 18.

In order to check whether my program works correctly, I will compute these parameters for the same paths manually.

2.3 Computing PDV and PVV Manually

2.3.1 PDV

The performed calculations are shown in Table 1.

The result obtained in the table is exactly the same as the result computed by my program.

2.3.2 PVV

The performed calculations are shown in Table 2.

The result obtained in the table is exactly the same as the result computed by my program.

Conclusion

According to the calculations and information specified in the task itself, the network I was suggested in this laboratory assignment complies with the standards.

In this laboratory work I familiarized myself with the ways to design an Ethernet network in the physical perspective. I learnt what are the standards governing the physical assembly of Ethernet networks and how

Table 1 – The manual calculation of PDV

N	Type	Position	Length (*m*)	Delay (*bt*)	Travel Time (*bt*)	Base (*bt*)	Total (*bt*)
1	C2	Left	150	0.1026	15.39	11.75	27.14
2	FO	Intermediate	900	0.1	90	29.0	119.0
3	FL	Intermediate	1200	0.1	120	33.5	153.5
4	C2	Intermediate	180	0.1026	18.468	46.5	64.968
5	C5	Right	400	0.0866	34.64	169.5	204.14
	Total		**2830**		**278.498**	**290.25**	**568.748**

Table 2 – The manual calculation of PVV

N	Type	Position	Variability (*bt*)
1	C5	Left	16
2	C2	Intermediate	11
3	FL	Intermediate	8
4	C5	Intermediate	11
5	T	Right	0
	Total		46

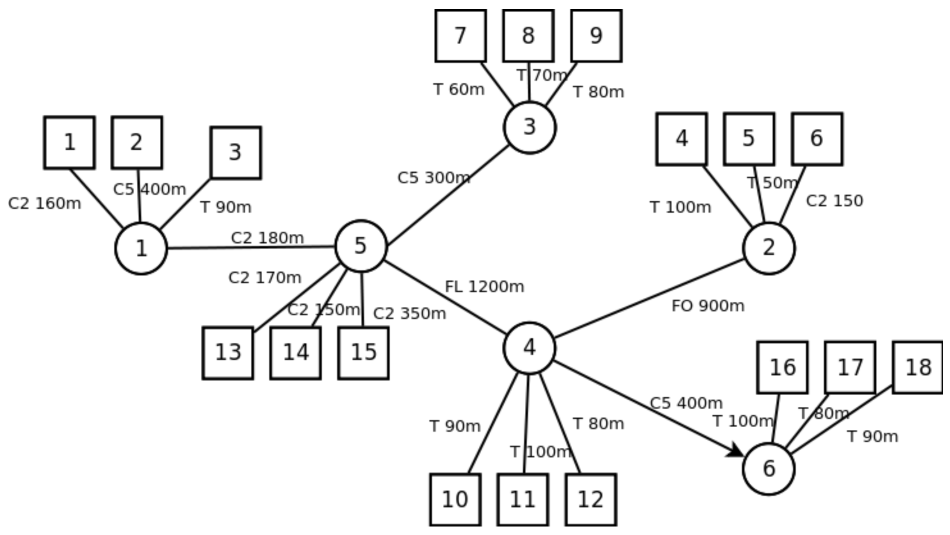

Figure 1 – The graphical representation of the suggested network

to comply with these standards. This knowledge is essential for any person who is ever going to work with Ethernet networks. I am strongly inclined to believe that even though I do not plan to become a network administrator, knowing how to compute the technical parameters of Ethernet networks will help me a lot.

References

1) http://www.bomara.com/Garrett/pdvnote.html
2) http://www.mark-itt.ru/collection/Ethernet/ethernet-config.html

Appendix A

Source Code

```haskell
{- lab1.hs

A helper program for lab1 in RC.
The following segment name abbreviations were adopted, according to
the suggestions in the task:
  * 10Base5         C5
  * 10Base2         C2
  * 10Base-T        T
  * 10Base-FB   FB
  * 10Base-FL   FL
  * 10Base-FP   FP
  * FOIRL           FO
  * AUI (>2m)   A

Input file format is:

number of edges in the concentrator graph
concentrator1 concentrator2 connectionType connectionLength
...

number of stations
station concentrator connectionType connectionLength
   ...
-}

import Data.Map (fromList, lookup)
import Data.Maybe (fromJust)
import Data.List (find, foldl')
import System.IO
import Text.Printf

-- An IEEE PDV specification entry.
data PDVSpec = PDVSpec {
      pdvBaseLeft          :: Double
    , pdvBaseIntermediate  :: Double
    , pdvBaseRight         :: Double
    , pdvSpecificDelay     :: Double
    } deriving (Show)

-- An IEEE PVV specification entry.
data PVVSpec = PVVSpec {
      pvvBaseLeft :: Double
    , pvvBaseIntermediate :: Double
    } deriving (Show)

type Node = Int
type LinkType = String
type Length = Double
type Weight = Double

-- The description of a link.
data Link = Link {
      linkStart :: Node
    , linkEnd :: Node
    , linkType :: LinkType
    , linkLength :: Length
    } deriving (Show, Eq)
```

```
-- Types of segments.
data SegmentType = LeftSegment | RightSegment | IntermediateSegment

-- The table of IEEE specifications for computing the PDV.
pdvSpecs = Data.Map.fromList ([
             ("C5", PDVSpec 11.75 46.5 169.5 0.0866)
            ,("C2", PDVSpec 11.75 46.5 169.5 0.1026)
            ,("T",  PDVSpec 15.25 42.0 165.0 0.113 )
            ,("FB", PDVSpec     0 24.0     0 0.1   )
            ,("FL", PDVSpec 12.25 33.5 156.5 0.1   )
            ,("FP", PDVSpec 11.25 61.0 183.5 0.1   )
            ,("FO", PDVSpec  7.75 29.0 152.0 0.1   )
            ,("A",  PDVSpec     0    0     0 0.1026)
            ])

-- The table of IEEE specifications for computing the PVV.
pvvSpecs = Data.Map.fromList ([
             ("C5", PVVSpec 16 11)
            ,("C2", PVVSpec 16 11)
            ,("T",  PVVSpec 10.5 8)
            ,("FL", PVVSpec 10.5 8)
            ,("FO", PVVSpec 10.5 8)
            ,("FB", PVVSpec 0 2)
            ,("FP", PVVSpec 11 8)
            ])

inputFilename = "input"
outputFilename = "output"

pdvBase :: SegmentType -> PDVSpec -> Double
pdvBase LeftSegment = pdvBaseLeft
pdvBase IntermediateSegment = pdvBaseIntermediate
pdvBase RightSegment = pdvBaseRight

pvvBase :: SegmentType -> PVVSpec -> Double
pvvBase LeftSegment spec = pvvBaseLeft spec
pvvBase IntermediateSegment spec = pvvBaseIntermediate spec
pvvBase RightSegment _ = 0

parseLink :: String -> Link
parseLink description =
    let parts = words description
    in Link (read (parts!!0)::Node) (read (parts!!1)::Node) (parts!!2)
                         (read (parts!!3)::Length)

parseNetwork :: [String] -> [Link]
parseNetwork list =
    let n = read (head list)::Int
    in map parseLink $ take n $ tail list

-- Adds the inverse of each edge to the central graph.
completeCentral :: [Link] -> [Link]
completeCentral = foldr step []
    where step oldLink@(Link node1 node2 linkType linkLength) links =
              oldLink:(Link node2 node1 linkType linkLength):links

pdvWeight :: SegmentType -> Link -> Double
pdvWeight segmentType link =
    let spec = fromJust $ Data.Map.lookup (linkType link) pdvSpecs
    in (pdvBase segmentType spec) + (linkLength link) * (pdvSpecificDelay spec)

pvvWeight :: SegmentType -> Link -> Double
```

```haskell
pvvWeight segmentType link =
    let spec = fromJust $ Data.Map.lookup (linkType link) pvvSpecs
    in pvvBase segmentType spec

listConcentrators :: [Link] -> [Node]
listConcentrators links = foldr step [] links
    where step current found =
              (checkAdd (linkStart current) . checkAdd (linkEnd current)) found
                  where checkAdd name list | elem name list = list
                                           | True = name:list

-- Produces only those links which start at the nodes from the first
-- list and end in the nodes of the second list.
filterLinks :: [Node] -> [Node] -> [Link] -> [Link]
filterLinks startFilter endFilter =
    filter (\n -> ((linkStart n) `elem` startFilter)
                 && (linkEnd n) `elem` endFilter)

-- Does a BFS starting at a concentrator and computes the weights of
-- all the paths it goes through.
exploreName :: Node -> [Node] -> [Link] -> (Link -> Weight) -> [(Weight, Node)]
exploreName name concentrators central weightFunc = (0, name) : explore name 0 [name]
    where explore name weight visited =
              let notVisited = filter (`notElem` visited) concentrators
                  toExplore = filterLinks [name] notVisited central
              in (weight,name) : foldr step [] toExplore
                     where step link result =
                               let end = linkEnd link
                                   linkWeight = weightFunc link
                               in (explore end (weight + linkWeight)
                                               (end:visited)) ++ result


-- Each of the resulting links will contain the total weight of the
-- path between the specified concentrators.
buildCentralWeights :: [Node] -> [Link] -> (Link -> Weight) -> [Link]
buildCentralWeights concentrators central weightFunc = foldr step [] concentrators
    where step name weights =
              let exploreResults = tail (exploreName name concentrators
                                                     central weightFunc)
                  newWeights = map buildLink exploreResults
              in newWeights ++ weights
                 where buildLink (weight, end) = Link name end "" weight

hostsByConcentrator :: Node -> [Link] -> [Link]
hostsByConcentrator name = filter (\link -> (linkEnd link) == name)

-- Evaluates the weight of the path between two hosts.
analyzeHostToHost :: Link -> Link -> Weight -> (SegmentType -> Link -> Weight) -> Weight
analyzeHostToHost leftLink rightLink intermediateWeight weightFunc =
    (weightFunc LeftSegment leftLink) + (weightFunc RightSegment rightLink)
                                      + intermediateWeight

-- Evaluates the weight of the path between all pairs of hosts.
analyzeAllPaths :: [Link] -> [Link] -> (SegmentType -> Link -> Weight) -> [Link]
analyzeAllPaths centralWeight peripheral weightFunc = foldr stepByConcentrators [] centralWeight
    where stepByConcentrators link weights =
              let leftHosts = hostsByConcentrator (linkStart link) peripheral
                  rightHosts = hostsByConcentrator (linkEnd link) peripheral
              in foldr (stepByLeft rightHosts) [] leftHosts ++ weights
                  where stepByLeft rightH left weights =
                            (foldr stepByRight [] rightH) ++ weights
```

```
                                 where stepByRight right weights =
                                          (Link (linkStart left) (linkStart right) "" (analyzeHostToHost
                                              left right (linkLength link) weightFunc)):weights

maxWeight :: [Link] -> Link
maxWeight links = foldr step (head links) links
    where step link found | (linkLength link) > (linkLength found) = link
                          | True = found

matrixOutput :: [Int] -> [Link] -> Handle -> IO ()
matrixOutput hosts weights outH = do
  hPutStr outH "  "
  mapM (\arg -> hPrintf outH "%8d" (arg::Int)) hosts
  hPutStrLn outH ""
  mapM outputRows hosts
  hPutStrLn outH ""
    where outputRows row = do
            hPrintf outH "%2d" (row::Int)
            mapM (outputCell row) hosts
            hPutStrLn outH ""
                where outputCell row column = do
                                    let link = filterLinks [row] [column] weights
                                    if link == []
                                      then (hPrintf outH "%8.3f" (0::Double))
                                      else (hPrintf outH "%8.3f" ((linkLength (head link))::Double))

main :: IO ()
main = do
  putStrLn "Lab1 in RC"

  raw <- readFile inputFilename
  let input = lines raw
      nCentralEdges = read (head input)::Int

      central = (completeCentral . parseNetwork) input
      peripheral = parseNetwork $ drop (nCentralEdges + 2) input

      concentrators = listConcentrators central

      centralPVV = buildCentralWeights concentrators central
                   (pvvWeight IntermediateSegment)
      centralPDV = buildCentralWeights concentrators central
                   (pdvWeight IntermediateSegment)

      pdv = analyzeAllPaths centralPDV peripheral pdvWeight
      pvv = analyzeAllPaths centralPVV peripheral pvvWeight

  outH <- openFile outputFilename WriteMode

  hPutStrLn outH "PDV Weight Matrix"
  -- This is a small bad ugly hack :-)
  matrixOutput [1..18] pdv outH

  let maxPDV = maxWeight pdv
  hPrintf outH "The maximal PDV: %8.3f (%d-%d).\n\n"
            ((linkLength maxPDV)::Double) ((linkStart maxPDV)::Node)
                                          ((linkEnd maxPDV)::Node)

  hPutStrLn outH "PVV Weight Matrix"
  matrixOutput [1..18] pvv outH

  let maxPVV = maxWeight pvv
```

```
hPrintf outH "The maximal PVV: %8.3f (%d-%d).\n\n"
          ((linkLength maxPVV)::Double) ((linkStart maxPVV)::Node)
                                        ((linkEnd maxPVV)::Node)

hClose outH
```