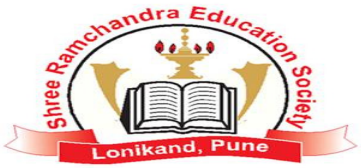
SHREE RAMCHANDRA COLLEGE OF ENGINEERING



Laboratory Manual

T.E. Computer Semester-VI

DEC 2015 –MAY 2016

Subject Code / Subject

(310255)

Programming Laboratory-IV

Course / Branch

Computer Engineering

(2014 course)

Executed by

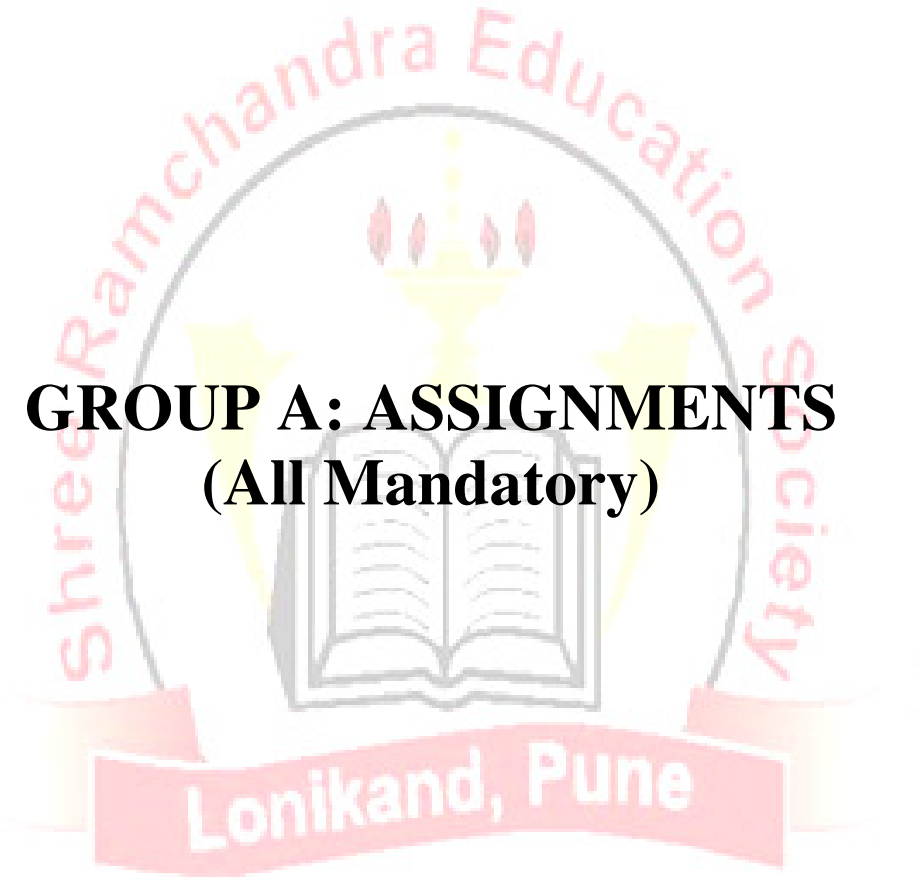
Prof. Sunil Deokule

Teaching Scheme:
Practical: 4 Hrs/Week

Examination Scheme:
Term Work: 50 Marks
Oral: 50 Marks

Prof. Sunil Deokule Thombre B.H Dr. Avinash Desai.

Staff in charge **H.O.D** **Principal**



GROUP A: ASSIGNMENTS
(All Mandatory)

Assignment No.	01
Title :	Implementation of Packet sniffer. Program should identify header of each protocol. Use multi-core programming
Roll No:	
Date:	

Assignment No:1

Title:
Write a program to Implement a packet sniffing tool in C++/Java/Python.
Aim: Study and Implementation of Packet Sniffer.
Prerequisites:
Basics of Networking, socket programming, etc.
Objectives:
- To learn the concept of packet sniffing
- To study the representation, implementation of Packet Sniffer.

Theory:
Packet Sniffer:
A packet sniffer (also known as a network analyzer, protocol analyzer or for particular types of networks, an Ethernet sniffer or wireless sniffer) is a computer program or a piece of computer hardware that can intercept and log traffic passing over a digital network or part of a network. As data streams flow across the network, the sniffer captures each packet and, if needed, decodes the packet's raw data, showing the values of various fields in the packet, and analyzes its content.

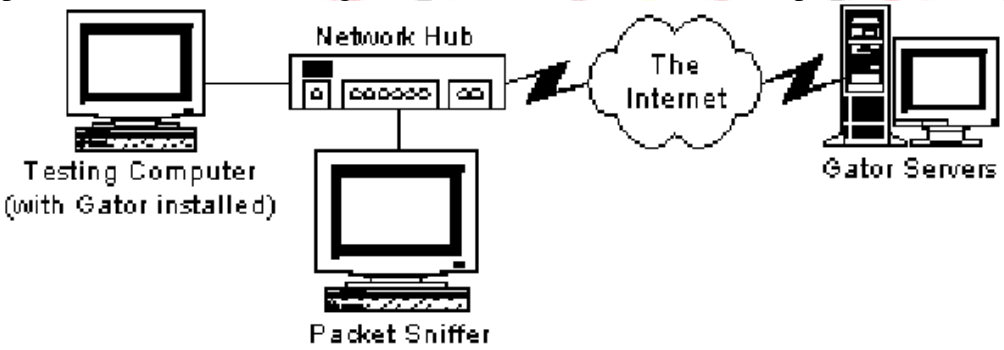


Figure 1:Block diagram for Packet Sniffer

Capabilities:
On wired broadcast LANs, depending on the network structure (hub or switch), one can capture traffic on all or just parts of the network from a single machine within the network; however, there are some methods to avoid traffic narrowing by switches to gain access to traffic from other systems on the network (e.g., ARP spoofing). For network monitoring purposes, it may also be desirable to monitor all data packets in a LAN by using a network switch with a so-called monitoring port, whose purpose is to mirror all packets passing through all ports of the switch when systems (computers) are connected to a switch port. To use a network tap is an even more reliable solution than to use a monitoring port, since taps are less likely to drop packets during high traffic load. On wireless LANs, one can capture traffic on a particular channel, or on several channels when using multiple adapters.
On wired broadcast and wireless LANs, to capture traffic other than unicast traffic sent to the machine running the sniffer software, multicast traffic sent to a multicast group to which that machine is listening, and broadcast traffic, the network adapter being used to capture the traffic must be put into promiscuous mode; some sniffers support this, others do not. On wireless LANs, even if the adapter is in promiscuous mode, packets not for the service set for which the adapter is configured will usually be ignored. To see those packets, the adapter must be in monitor mode.When traffic is captured, either the entire contents of packets can be recorded, or

the headers can berecorded without recording the total content of the packet. This can reduce storage requirements, and avoid legal problems, but yet have enough data to reveal the essential information required for problem diagnosis. The captured information is decoded from raw digital form into a human-readable format that permits users of the protocol analyzer to easily review the exchanged information. Protocol analyzers vary in their abilities to display data in multiple views, automatically detect errors, determine the root causes of errors, generate timing diagrams, reconstruct TCP and UDP data streams, etc. Some protocol analyzers can also generate traffic and thus act as the reference device; these can act as protocol testers. Such testers generate protocol-correct traffic for functional testing, and may also have the ability to deliberately introduce errors to test for the DUT's ability to deal with error conditions. Protocol analyzers can also be hardware-based, either in probe format or, as is increasingly more common, combined with a disk array. These devices record packets (or a slice of the packet) to a disk array. This allows historical forensic analysis of packets without the users having to recreate any fault.

Uses:

The versatility of packet sniffers means they can be used to:

- Analyze network problems
- Detect network intrusion attempts
- Detect network misuse by internal and external users
- Documenting regulatory compliance through logging all perimeter and endpoint traffic
- Gain information for effecting a network intrusion
- Isolate exploited systems
- Monitor WAN bandwidth utilization
- Monitor network usage (including internal and external users and systems)
- Monitor data-in-motion
- Monitor WAN and endpoint security status
- Gather and report network statistics
- Filter suspect content from network traffic
- Serve as primary data source for day-to-day network monitoring and management
- Spy on other network users and collect sensitive information such as login details or users cookies (depending on any content encryption methods that may be in use)
- Reverse engineer proprietary protocols used over the network
- Debug client/server communications
- Debug network protocol implementations
- Verify adds, moves and changes
- Verify internal control system effectiveness (firewalls, access control, Web filter, spam filter, proxy) Packet capture can be used to fulfill a warrant from a law enforcement agency (LEA) to produce all network traffic generated by an individual. Internet service providers and VoIP providers in the United States must comply with CALEA (Communications Assistance for Law Enforcement Act) regulations. Using packet capture and storage, telecommunications carriers can provide the legally required secure and separate access to targeted network traffic and are able to use the same device for internal security purposes. Collection of data from a carrier system without a warrant is illegal due to laws about interception.

Capturing Packets with libpcap:

All data on the network travels in the form of packets, which is the data unit for the network. To

understand the data a packet contains, we need to understand the protocol hierarchy in the reference models. The network layer is where the term packet is used for the first time. Common protocols at this layer are IP (Internet Protocol), ICMP (Internet Control Message Protocol), IGMP (Internet Group Management Protocol) and IPsec (a protocol suite for securing IP). The transport layer's protocols include TCP (Transmission Control Protocol), a connection-oriented protocol; UDP (User Datagram Protocol), a connection-less protocol; and SCTP (Stream Control Transmission Protocol), which has features of both TCP and UDP. The application layer has many protocols that are commonly used, like HTTP, FTP, IMAP, SMTP and more. Capturing packets means collecting data being transmitted on the network. Every time a network card receives an Ethernet frame, it checks if its destination MAC address matches its own. If it does, it generates an interrupt request. The routine that handles this interrupt is the network card's driver; it copies the data from the card buffer to kernel space, then checks the ethertype field of the Ethernet header to determine the type of the packet, and passes it to the appropriate handler in the protocol stack. The data is passed up the layers until it reaches the user-space application, which consumes it. When we are sniffing packets, the network driver also sends a copy of each received packet to the packet filter. To sniff packets, we will use libpcap, an open source library.

Understanding libpcap

libpcap is a platform-independent open source library to capture packets (the Windows version is winpcap). Famous sniffers like tcpdump and Wireshark make the use of this library. To write our packet-capturing program, we need a network interface on which to listen. We can specify this device, or use a function which libpcap provides: `char *pcap_lookupdev(char *errbuf)`. This returns a pointer to a string containing the name of the first network device suitable for packet capture; on error, it returns NULL (like other libpcap functions). The `errbuf` is a user-supplied buffer for storing an error message in case of an error — it is very useful for debugging your program. This buffer must be able to hold at least `PCAP_ERRBUF_SIZE` (currently 256) bytes.

Getting control of the Network Device

Next, we open the chosen network device using the function

`pcap_t *pcap_open_live(const char *device, int snaplen, int promisc, int to_ms, char *errbuf).` It returns an interface handler of type `pcap_t`, which other libpcap functions will use.The first argument is the network interface we want to open; the second is the maximum number of bytes to capture. Setting it to a low value will be useful when we only want to grab packet headers. The Ethernet frame size is 1518 bytes. A value of 65535 will be enough to hold any packet from any network. The `promisc` flag indicates whether the network interface should be put into promiscuous mode or not. (In promiscuous mode, the NIC will pass all frames it receives to the CPU, instead of just those addressed to the NIC's MAC address.) The `to_ms` option tells the kernel to wait for a particular number of milliseconds before copying information from kernel space to user space. A value of zero will cause the read operation to wait until enough packets are collected. To save extra overhead in copying from kernel space to user space, we set this value according to the volume of network traffic.

Actual capture

Now, we need to start getting packets. Let's use

`u_char *pcap_next(pcap_t *p, struct pcap_pkthdr *h).`

Here, `*p` is the pointer returned by `pcap_open_live();` the other argument is a pointer to a variable of type `struct pcap_pkthdr` in which the first packet that arrives is returned.

The function **int pcap_loop(pcap_t *p, int cnt, pcap_handler callback, u_char *user**) is used to collect the packets and process them. It will return when cnt number of packets have been captured. A callback function is used to handle captured packets (we need to define this callback function). To pass extra information to this function, we use the *user parameter, which is a pointer to a u_char variable (we will have to cast it ourselves, according to our needs in the callback function).
The callback function signature should be of the form:
void callback_function(u_char *arg, const struct pcap_pkthdr* pkthdr, const u_char* packet).

The first argument is the *user parameter we passed to pcap_loop(); the next argument is a pointer to a structure that contains information about the captured packet. The structure of struct pcap_pkthdr is as follows (from pcap.h):
struct pcap_pkthdr {
struct timeval ts; /* time stamp */
bpf_u_int32 caplen; /* length of portion present */
bpf_u_int32 len; /* length of this packet (off wire)
*/
};
An alternative to pcap_loop() is pcap_dispatch(pcap_t *p, int cnt, pcap_handler callback, u_char *user). The only difference is that it returns when the timeout specified in pcap_open_live() is exceeded.
Filtering traffic
Until now, we have been just getting all the packets coming to the interface. Now, we'll use a pcap function that allows us to filter the traffic coming to a specific port. We might use this to only process packets of a specific protocol, like ARP or FTP traffic, for example. First, we have to compile the filter using the following code:
int pcap_compile(pcap_t *p, struct bpf_program *fp, const char *str, int optimize, bpf_u_int32 mask);
The first argument is the same as before; the second is a pointer that will store the compiled version of the filter. The next is the expression for the filter. This expression can be a protocol name like ARP, IP, TCP, UDP, etc. You can see a lot of sample expressions in the pcap-filter or tcpdump man pages, which should be installed on your system. The next argument indicates whether to optimize or not (0 is false, 1 is true). Then comes the netmask of the network the filter applies to. The function returns -1 on error (if it detects an error in the expression).After compiling, let's apply the filter using int pcap_setfilter(pcap_t *p, struct bpf_program *fp). The second argument is the compiled version of the expression.
Finding IPv4 information
int pcap_lookupnet(const char *device, bpf_u_int32 *netp, bpf_u_int32 *maskp, char *errbuf)
We use this function to find the IPv4 network address and the netmask associated with the device. The address will be returned in *netp and the mask in *mask.
Mathematical Model
Conclusion:
Hence, we have successfully studied concept of Packet Sniffer

GROUP C: ASSIGNMENTS

(All Mandatory)

Assignment No.	01
Title :	configuration of Dual stack IPv6 and IPv4 network
Roll No:	
Date:	

Assignment No:1

Title:
configuration of Dual stack IPv6 and IPv4 network
Aim: configuration of Dual stack IPv6 and IPv4 network
Prerequisites:
Basics of Networking, etc.
Objectives:
- To learn the concept of IPv6 and IPv4
- To study the representation, implementation of IPv6 and IPv4.

Theory:
Doing an IPv6 implementation project does not involve tearing down an aging IPv4 network and replacing it with a new IPv6-enabled network. Instead, the IPv4 and IPv6 networks will run in parallel in what the industry calls a "dual-stack" network. But IPv4 and IPv6 are so significantly different in design that network management toolsdesigned for an IPv4 network may not work the same in an IPv6 environment.

Doing an IPv6 implementation project does not involve tearing down an aging IPv4 network and replacing it with a new IPv6-enabled network. Instead, the IPv4 and IPv6 networks will run in parallel in what the industry calls a "dual-stack" network. But IPv4 and IPv6 are so significantly different in design that network management tools designed for an IPv4 network may not work the same in an IPv6 environment.

In this second installment of a three-part series on IPv6 implementation, Network Computing looks at the issues involved in deploying an IPv6 network alongside an IPv4 network.

The IPv6 protocol was established because the number of IPv4 addresses is quickly running out. The IPv6 protocol creates a 128-bit address, four times the size of the 32-bit IPv4 standard, so there will be infinitely more available IP addresses. This will accommodate all the smartphones, tablets and other computers on the network, but also the coming proliferation of Internet-connected devices including refrigerators, cars, and myriad sensors in homes, buildings and on IP networks.

With IPv6, a company may have exponentially more Internet addresses to use, but also more to manage, says Leslie Daigle, chief Internet technology officer for the Internet Society (ISOC), a global nonprofit organization that certifies technical standards for the Internet.

"The IPv6 address space is so large and your allocation is likely to be larger than you need it to be," she says. "On the flip side, that makes it a lot harder to probe your entire network because it is a much larger space."

The volume of available IP addresses adds to the network operator's workload because they have to probe the "dark spaces" within the network where there are no assigned IP addresses. "The managing and making sure that no one is squatting in your address space is considered to be a possible additional challenge," says Daigle.

The ISOC has created a Web portal, Deploy 360, to share information about how to deploy an IPv6-compliant network. On the site are a number of case studies on how IPv6 rollouts went, including one about the project at Oxford University in the United Kingdom. In an online report, Oxford's Guy Edwards detailed a five-step plan for deploying IPv6 alongside the existing IPv4 network.

First, Edwards advises, the organization should perform a network device audit, identifying all the routers, switches and firewalls on the network, as well as what specific versions of hardware and software are running. With the help of networking vendors, the next step is to determine which of the devices are already IPv6-compliant. He also advises that network administrators run a test on a particular IPv6 device to make sure that the software application to run on the network works.

Dual Stack IPv4/IPv6

When a device has dual stack capabilities then it has access to both IPv4 and IPv6 technology available. It can use both of these technologies to connect to remote servers and destinations in parallel.

How does a device know when to use IPv4 or IPv6?

When a client wants to connect to a server (e.g., www.example.com), the client issues two DNS requests in parallel: one request for IPv4 addresses, and one request for IPv6 addresses. After receiving the responses, the client generally follows the process described in IETF standard RFC3484 "Default Address Selection for IPv6" which leans towards an assumption that dual stack is a state of transitioning towards and IPv6-only network, and hence prefers IPv6 above IPv4 by design. When a client receives a response including both an IPv4 and IPv6 address then based upon RFC3484 the IPv6 address is the preferred address. If, for whatever reason, the usage of that address was non-successful, an alternate address will be used, potentially a valid IPv4 address to connect to the remote location.

What if I run dual stack and I want to connect to a single stack IPv4 or IPv6 remote site?

It is possible for a dual stack device to connect to an IPv4-only or IPv6-only device. The protocol selected, IPv4 or IPv6 is based upon the IP addresses received within a DNS response. If the DNS server returned only IPv4 addresses, then IPv4 is used, if the DNS server returned only IPv6 addresses then IPv6 is used.

How does DNS work with IPv4 and IPv6 addresses?

There are two elements with respect to DNS. The first is related to the protocol the DNS client is using to speak to the DNS server, which could be IPv4 or IPv6 based. Most DNS servers support both IPv4 and IPv6 initiated requests.

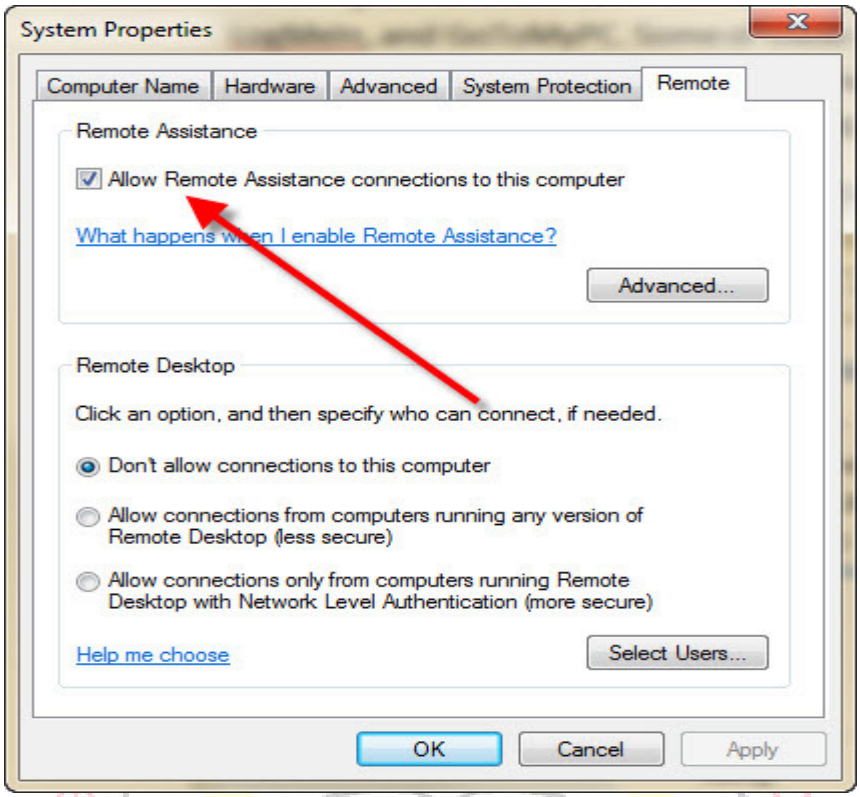
Assignment No.	02
Title :	configuration of Dual stack IPv6 and IPv4 network
Roll No:	
Date:	

Assignment No:2

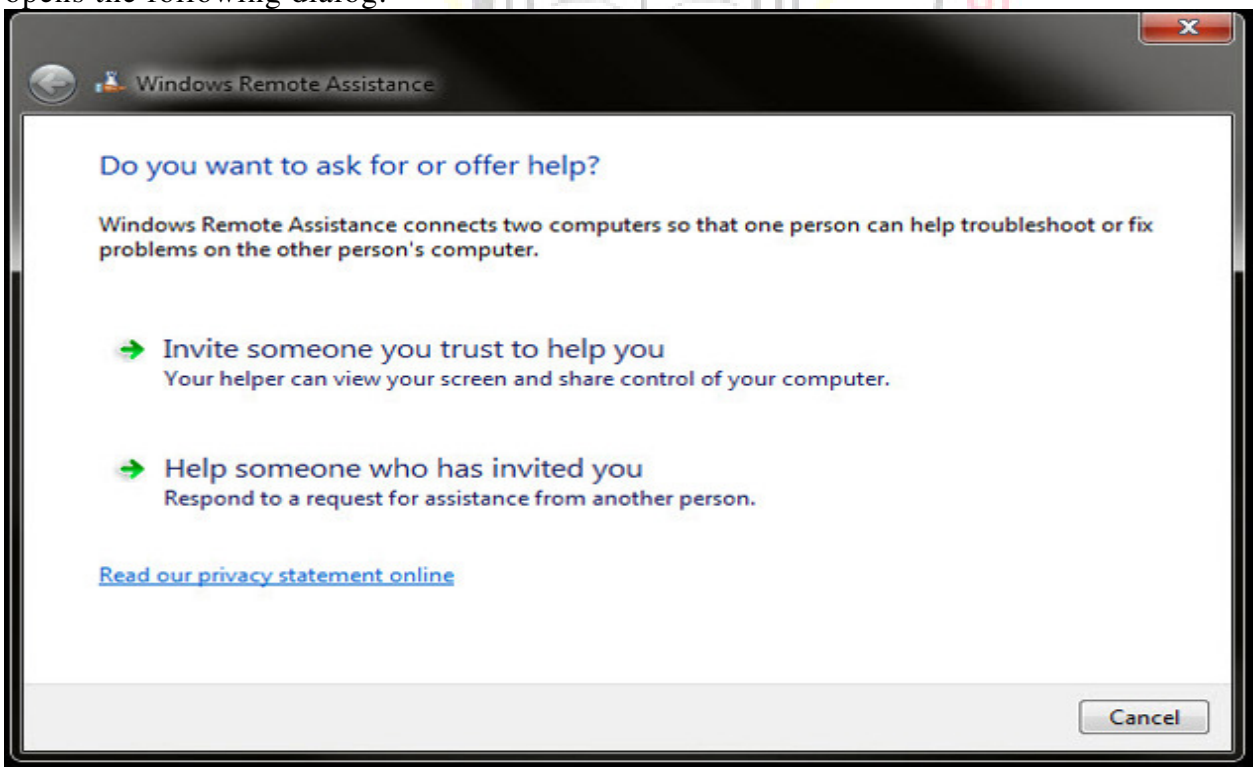
Prerequisites for Connecting to a PC using Remote Connection Since most Windows users are still on Windows 7, I'll go through the process in that OS. Pretty much the same process works for Vista (in case you're still using that much-derided OS), and in Windows 8. Windows 8 actually offers a new-style (formerly known as "Metro") app for remote desktop connections, too, and it works on Windows RT as well as Windows 8 and Windows 8 Pro.

1. Make sure both PCs are powered up and connected to the Internet. They can't be in Sleep or Hibernate state, either. To prepare the "host," or the machine that you'll be taking control of

2. Enable Remote Assistance. Open the Control Panel, and type "Remote" in its search box. You could also right-click on Computer and choose Properties, and then choose Remote settings on the left panel. You'll open a Properties sheet with the top choice of "Allow Remote Assistance connections to this computer." Make sure this box is checked.

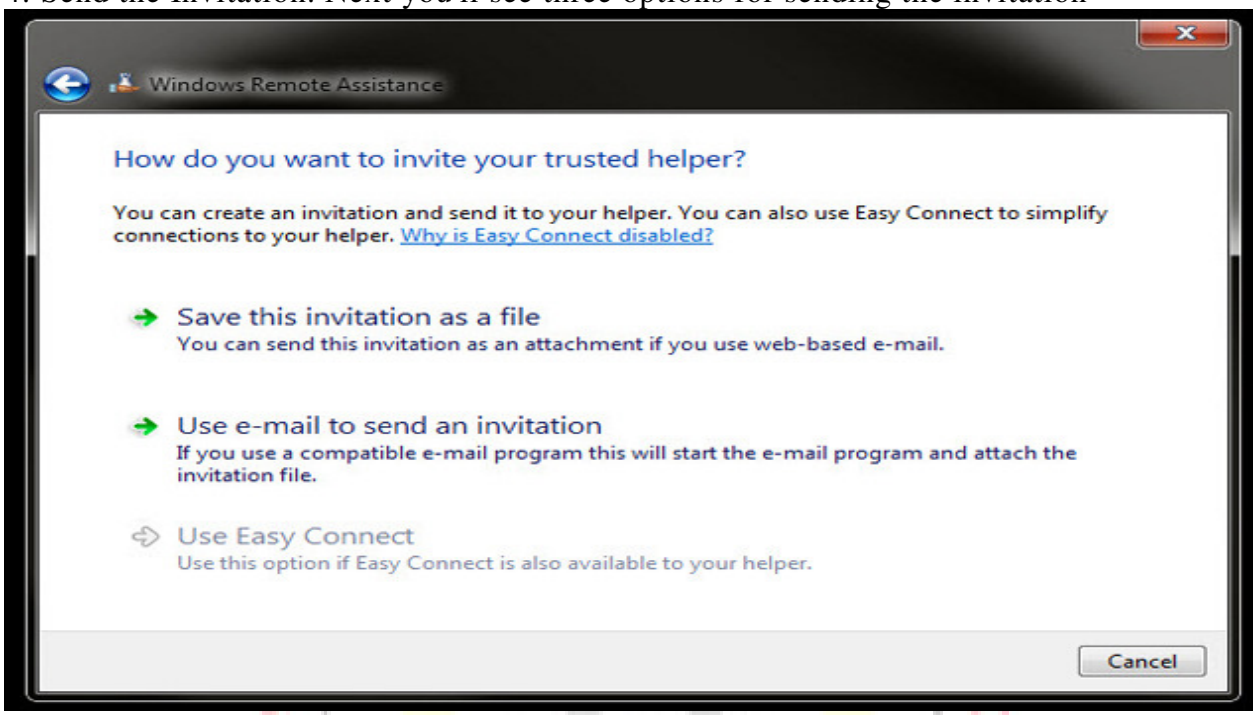


3. Ask someone to connect. At the computer to be controlled, type "Remote Assistance" in the Start button's search box, and then click on Windows Remote Assistance. This opens the following dialog:



Click "Invite someone you trust to help you."

4. Send the Invitation. Next you'll see three options for sending the invitation



You'll notice that the last (and best) option, Use Easy Connect, is grayed out on my screenshot: This will be the case if both computers aren't using Windows 7 or 8, with some corporate networks, and if your router doesn't support Peer Name Resolution Protocol. When I connected to a public Wi-Fi network, the option became available. In any case, send the invitation to the user of the computer that's going to do the remote controlling. And Easy Connect lives up to its name: If it's available, that's the one you should use.