|  |
| --- |
| ECE 4220 Real Time Embedded Computing |
| Low-Impact Network Custodian |
| Spring 2014 |
|  |
| **Kalen Brown** |
| **5/9/2014** |

|  |
| --- |
|  |

Introduction

The proposed project is a method to detect the status of a computer network based on packets sent across the network at a known period. This project uses sockets to send a one-way trip UDP packet from a TS-7250 board through an existing router and into a receiving TS-7250 board. It allows for smaller and fewer packets than a TCP Ping command and also allows the packet to be dropped completely. This allows the receiving board to know if there is an increase or decrease in network speed, if the router is overloaded causing occasional dropped packets, or if the network is down. The project will use a relay to trigger a hard reset of the network systems power supply.

This project will parallel one computer trying to reach another across a local area network using Internet Protocol (IP). The tests will use the User Datagram Protocol (UDP) with smaller packets than TCP that also do not allow for retransmission or ICMP messages to reduce the impact on the network. UDP is also used in real-time networking to reduce congestion and drop packets without retransmission to allow the most current packets the most bandwidth. In the simplest case, the home network has only one router. The receiving board will test for speed changes, an overloaded router, and a downed network. Both boards will be a TS-7250 with a real-time operating system. They each have a hardwired network interface and a breakout board with LEDs, buttons, and a speaker. Both boards will use the network interface, but only the receiving board will use the breakout board. If any of the LEDs on the receiving board are lit, then the power status of the router is on. The green, yellow, and red LEDs indicate a recent increase, no change, or decrease in the network speed respectively. An alternating flashing of the yellow and red LED means that some of the packets were lost recently. This indicates that the router was recently overloaded. When no LEDs are lit and the speaker is sounding a tone, this indicates that the router is being reset in an attempt to restore the network.

Goals and Objectives

The goal is to help general public users more easily understand the status of their network and help facilitate in the automatic reset of the system in a catastrophic failure. People use their home network on regular basis. They expect a minimum speed throughout the day and become frustrated when that changes without their knowing the cause. It is hard for people to tell if a router is overloaded or if the network is just experiencing a decrease in speed. Often times, the user needs to perform hard reset regardless of the cause of a network shutdown. Network engineers can monitor the status of a network using the Ping command, but this requires two-way TCP communication and many packets to flood the network. Routers on a network perform many functions and can become misinformed or corrupt as to the status of the network. A router itself may think that it is set up correctly and is just not currently receiving traffic, even though the numbers it is trying to use are out of date. Usually the solution is solved by the user via the reset button on the router. This reset button triggers the purging of old data and the construction of new data. It also gets new IP addresses from the modem and gives new IP addresses to the LAN computers. There are not commercially available third-party systems that can monitor the traffic through a router and act if the packets cannot traverse the network. There are applications that can show the current speed of the network, but they require many packets and two way communication that may flood the network. Some users set up a timer to reset their router on a regular 24 hour period, but this adds the potential for the reset time to be undesired and potentially during primetime use hours if the clock is incorrect. These are known problems for which there is no commercially standard solution and which this project seeks to correct. The primary user of the proposed project will be the general public or small businesses.

Standards

This project uses the Open Systems Interconnection (OSI) model networking standards to facilitate the transmission and reception of a network packet. This project uses Ethernet, sockets, routers, Internet Protocol version 4 using User Datagram Packets. The Linux operating system with a real time patch, and a TS-7250 board is used for both the sender and receiver boards.

Assumptions

The project does not have to discern a disconnected cable from a downed network, as a port may have actually gone down on the router and has to be rectified. It is assumed that a hard reset via a power cycle will not damage the router in any way, shape, or form. It is assumed that the network itself can recover from power cycling all the network system's devices at a random time. For testing purposes, the time to power cycle and the time between power cycles may be short. There can be multiple routers and modems on the network, but the system will only test routers that are a possible route between the sender and receiver. The IP address of the receiver must be known unless a viable way can be made to broadcast over the network and find the receiving node.

Proposed implementation

The sending board sets up a socket connection to the receiver using UDP as its protocol. It sets up a periodic real time task as its primary function and sets up a watchdog timer that is reset on every period. It then starts an iterator counter at zero as the virtual sequence number for the packets which increment on every period. This primary function is triggered on every timer period to send a UDP packet with the virtual sequence number, the period in nanoseconds, and the time of day. The size of the packet must be forever constant to ensure that the time to traverse the network is not impacted by a changing packet size. For testing purposes, the LEDs or the speaker may cycle to indicate the sending of a packet. The period of the real time task is to be determined, but should be 1 second or greater to minimize impact on the network. It is imperative that the sending board use a real time periodic task because if the sending of the packet fluctuates in the least, the receiving board assumes that it was caused by the network and yields false speed change indications. If the delay of sending the packet is longer than an entire period, i.e. missing the periodic task, then the receiver initially assumes the packet has been dropped. If the packet is later received, the receiver then assumes a large decrease in the network speed. If the sending unit has completely stalled and is never going to recover, the receiving unit assumes that the network is down and continues to cycle the network power until either a threshold is reached or something causes the sending unit to reset. This reveals the purpose of the watchdog timer: resetting the sending board.

The receiving board does all of the hard work. It receives the packets at an unknown time, and it stores the arrival time and the packet payload as sent from the socket. It examines the payload for the sequence number and stores it in a corresponding buffer location that holds approximately 1 minute of data for testing purposes. It continues this process for every packet received. In the event that two sequential numbers are received, the receiving board then calculates the difference in the time received between the sequential packets and compares that to the expected period. In the case of equal time difference and period, then the network speed has no change. If the time difference is less than the expected period, then an increase in speed is stored. If the time difference is greater than the expected period, then a decrease in speed is stored. These cases take care of a nonsequential packet by marking one as a slowdown and the other as a speed up. In the case of a packet sequence number that has been expected but missing for a packet loss time threshold, the packet is marked as a loss and the router is recorded as overloaded. This initial threshold is set to 2 seconds, but testing will verify the best value. In the case of many dropped packets and no new received packets, the network is determined to be down and the network equipment is reset. The time to reset the network is set via the SO\_RCVTIMEO argument to the socket setup. If the read function returns without receiving any data, the network is considered to be down.

Actual Implementation

The final implementation used UDP sockets, binding ports to processes, initial broadcasting, temporary broadcasting, regular broadcasting, real time periodic tasks, system calls, file operations, and other topics discussed in class. Occasionally, a timeout was set for a blocking function such as a socket read (socket(7), 2014). I also used a timeout check for data in the buffer from the user or if the program needed to shut down (NG, 2010). This allowed for provisional polling where it would break out of the blocking function as soon as the data is available or there is a timeout. The use of this and the time to time out is usually determined by experimentation. Application layer implementation of pseudo-TCP was implemented because there was retransmission of packets if they were anticipated as lost during the initialization phase. The UDP packet does not have a sequence number built in. TCP also has a sequence number which was implemented to keep track of nonsequential, duplicated, and dropped packets. This received packet tracking system became very large in its logic and memory size. It would be constantly updated and checked for old flows. These would require a linear search because of the circular buffer due to infinite packets. This would have been implemented with an incoming processing packet thread and another thread to iterate the array every period. This tracking system was abandoned for a more on-the-fly approach where the system checks the socket with a timeout where the timeout is equal to two periods. The originally proposed tracking system required many logic checks and was made a lower priority since it didn't use any topics used in the class and lab.

The final implementation uses a real time periodic task in user space to send the socket packets. The period was hard coded with the hopes to later implement a way to receive a new period from the client. There is no kernel module in the transmitting board because it is essentially a network node and nothing else. The client-to-server interaction became much more intricate than anticipated so it has an application layer link establishment. When I start the server before the client, the server waits for a message to come over the socket. Starting up the client initially sends a broadcast message looking for the server. The server hears this message, verifies that it matches our key for an initialize message, sends an acknowledgement message, and starts sending packets. The other side of the test is to start the client before the server, which also accounts for the situation where the UDP packet that initializes the server is dropped. In this case, the client sends the broadcast message but never receives a reply. It keeps trying to receive until a timeout occurs, and then it rebroadcasts the initializing message. This repeats until either a reply from the server is found, actual data from the server is found, or the user quits the program properly by typing the 'Q' or 'q' command as will be mentioned later. The client must also switch between bound and unbound mode on the socket during this startup time because it cannot broadcast before a message is received and the socket has been bound to.

When a packet is received, the system acquire a timestamp with the “get time of day” function to record the time it was read into the log file. The contents of the packet are also stored in the log file for later reference by the user. If the timeout is reached on the socket read, the packet is marked as dropped, and it will try to receive again. After no packets have been received for a set threshold, then the system attempts to reboot the router by triggering the relay and waiting for the router to shut down. The shutdown time is short, but the booting time is very long so it must be plenty of time to boot up. This reboot is attempted a set number of times (in testing, it was three). When the router is being cycled, a kernel is used to make a more discernible periodic sound on the speaker and trip the router relay. The user space initiates this task, and also disables it. After three attempts, if the packets are still not being received the program initiates the shutdown procedure to prevent further power cycling of the router.

Hardware implementation involved the GPIO port B of the TS-7250 board to trigger a prebuilt relay. The relay used an external power supply of 9 volts generated with a generic switchable voltage power supply initially and later used a 9 volt battery just to allow that power supply for to be used by the router. The switchable power supply was then set to 12 volts, the hot pin of the output was run through the normally closed section of the relay, and the hot pin came from the relay to the router. The ground of the router is kept isolated from the grounds for the board and the relay. Each device has its own isolated electrical ground. The final diagram of the project is shown in figure 1.

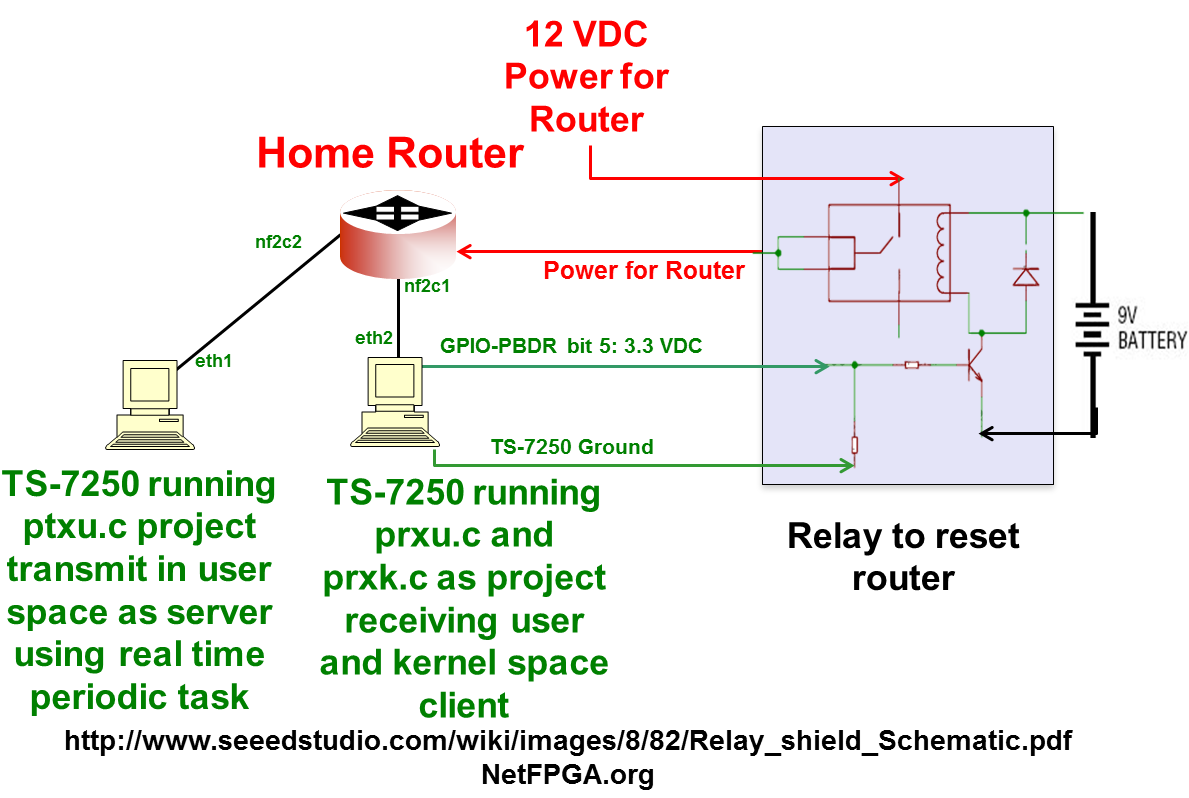


Figure 1 Project Diagram

Experiments and results

For the tests, the router easily went between the sending board and the network that linked it to the rest of the boards; however, only put boards that were not running my client could be on the router’s network because the reset function would halt the program and prevent the router from turning back on. When the router did reboot, the sending boards pulled their next instruction and picked up where they left off, thereby signaling my client board that the reset did in fact work. The router used was a simple Netgear Wi-Fi router that was actually out of service because of the exact problems identified in the goals of this project.

The initial test is to establish an application layer link between the two boards using sockets. Since UDP is being used, we have to implement our own error checking. This was shown by starting up the server and the client, then letting them find each other. The process was described in the implementation section, but the process and verification of communication can be shown in the log and on the screen in debug mode. There were many different scenarios to test, and the list kept growing. Primarily I tested for starting the client before the server and starting the server before the client. These tests proved very reliable on both sides. The only issue found was if the server was already in the transmitting state, and the client started up, then the client had to search through many of the packets in the socket until it hopefully found the acknowledgement that the server received the initialize function. Sometimes this message was lost due to buffer overrun. This was impossible to show because it was just a loop of received messages from the server without the acknowledgement in it.

There were also error checking tests because a board cannot bind two programs to the same socket, or even the same program to the same socket for another thread. This is done by starting up two clients or servers on the same port and on the same board and allowing them to reach the binding point. The first process to request the bind achieves the bind. The second process to request the bind is denied and will properly exit the program.

Proper shutdown was a large part of the program as well. If the file was opened but the program was closed by the user before closing the file, the file could be left open or may lead to a corrupted write to the file. This was tested for and the problem is shown in the log file:

"

rx'd on 1399605411.517312    Testing Messages Testing Messages

rx'd on 1399605412.016917    Testing Messages Testing starting a new log entry

rx'd on 1399606638.357152    Testing Messages Testing Messages

"

Also the server will keep sending unnecessary messages over the network even though the client has been shut down. To prevent these things, the entire system must know when to shut down everything. First we startup using one of the methods from the previous paragraph, once an application link is established we can issue the shutdown command. This is performed by the user hitting "Q" or "q" on the receiving board. This sets a flag to the other threads that are running that they need to shut down when they are done processing data. It also sends a non-broadcast message to the server telling it to stop broadcasting and end its program as well.

The other test where I actually cause the router to overload and start dropping packets or completely shut down would have involved a large amount of programs in an infinite loop of broadcasts. They don't need to be on the same port, but they do have to be on the same network as the network under test. I had to hook two extra boards into the router, connected one port to the Mizzou network, and another to the sending board. I initiated the attack, but nothing out of the ordinary happened. This test was a failure because I do not have the resources necessary to cause an overload on this particular router. The partial dropping of packets is a rare event, but it does occur and is a problem in general and should be accounted for. An attempt was made to flood the network with many boards running an infinite loop of broadcasts on many different ports. I was not able to find a way to move enough data to overload the router. To prove my concept I would have needed to succeed this test and observe packets being dropped without my intervention in hardware or changing settings to cause the drop.

As a workaround to this failure, I can simulate some packets dropped by pulling the cable and then quickly attaching the cable before the client tries to reset the router. This test was proven to work and packets were recorded as dropped in the log as shown below. It started to reset the router, but the router was shown to recover before the subsequent reset cycles.

"rx'd on 1399613516.098415    Testing Messages Testing Messages

rx'd on 1399613516.597962    Testing Messages Testing Messages

number of dropped packets this round == 1

number of dropped packets this round == 2

number of dropped packets this round == 3

number of dropped packets this round == 4

number of dropped packets this round == 5

number of dropped packets this round == 6

number of dropped packets this round == 7

resetting router!!! Attempt #1

rx'd on 1399613633.406500    Testing Messages Testing Messages

rx'd on 1399613633.407426    Testing Messages Testing Messages

rx'd on 1399613633.408127    Testing Messages Testing Messages

rx'd on 1399613633.408783    Testing Messages Testing Messages"

Total dropped packets can be simulated in the same way, without reconnecting the cable before the total reset. This test also shows that after a few resets, we decide that we aren't doing any good and we shut down the system. The relay did trip the router power supply and allow the router to reboot properly. The program initiated the soft shut down after the thresholds were reached.

"rx'd on 1399609101.164559    Testing Messages Testing Messages  
rx'd on 1399609101.664000    Testing Messages Testing Messages  
rx'd on 1399609102.163515    Testing Messages Testing Messages  
rx'd on 1399609102.663198    Testing Messages Testing Messages  
number of dropped packets this round == 1  
number of dropped packets this round == 2  
number of dropped packets this round == 3  
number of dropped packets this round == 4  
number of dropped packets this round == 5  
number of dropped packets this round == 6  
number of dropped packets this round == 7

resetting router!!! Attempt #1

number of dropped packets this round == 8

resetting router!!! Attempt #2

number of dropped packets this round == 9

resetting router!!! Attempt #3

Program not properly resetting the system.

Shutting down to prevent damage."

Finally, testing the increase and decrease in speed and being sure that a packet in the sequence is being dropped automatically has failed due to not being implemented. The tests for the file writing, setting up the connection, and dealing with total loss of packets by reporting it and resetting the router if necessary were a success. This shows that the primary issue of having to reset the router manually will be solved by this implementation.

Discussion

Originally I wanted to help make the sending of the packet as periodic as possible. I knew I needed a real time process and I knew that after I requested the packet to be sent that it would enter the network queue thereby entering the network realm under test. I knew that I didn't want to use the timer interrupt because this would induce drift and the timer would have to be reset every time which would create an increasing drift in the trigger time. Ideally I would put this task in the kernel and disable all interrupts upon the period trigger to prevent any preemption if possible, then enabling all interrupts that were previously disabled. The issues began when I researched and found that the sockets exist in user space memory which by definition cannot be directly accessed from kernel space without some kind of bridge. Even with some kind of bridge like a pipe, you cannot randomly access the data requested. Through a lot of researching, I found programs that either required many header files like netpoll or simply didn't appear to work (Gawenda). I finally found a program that would send out a UDP message from kernel space from (Garcia-Navarro, 2006). This program successfully sent the packets after I performed my own flipping of the byte order for the port and the IP address because HTONS functions were not available in the kernel space. Not having the functionality to check what type of system we are on greatly decreases the portability of the program. Also the program did not have broadcast initially and I was only implementing the sending of the packet, so it only will send packets to a specific IP address and port number after I edited it. This program caused a kernel panic at random times which caused me to implement my watchdog timer because I thought it was caused by something that I was doing incorrectly. The watchdog timer would destroy the hung task and restart it. This appeared to reset the kernel panic but since a kernel panic can also mean other things are or may be corrupted. While this was a very interesting learning experience, I decided to abandon the kernel task because partially locking up the system is obviously not desired and the methods being used to send the packets were very unorthodox and not recommended.

In the process of testing this kernel, I reduced my real time periodic task to a do get time of day call and a print k statement and realized that the call was not actually real time. I observed this in the user space real time task as well. I attempted to use other periods and talked to the TA about this and he said that he has observed this problem in the past in a particular instance. observed a "drift" in doing this both in kernel space and use space. This was verified by other classmates in their programs output and by the TA. We could not fix the drift but we did observe that the drift appeared to attempt to correct itself after becoming too late by starting the next call a little early. It didn't appear to actually be able to correct itself correctly. This ruined the concept of my program since the drift was inducing a false speedup or slowdown of the network into my program. Also it was observed that if only viewing the seconds you would actually miss a second and cause the incrementing option by the periods to skip a sequence. As in if the period was one second (and many periods were tried) then after initialization we should see 1, 2, 3, 4, 5, 6, 7, 8, 9... but eventually we observed 20, 21, 23, 24, 26 with this skipping pattern repeating over the data set with the period of the skip apparently changing the with actual period. A rough average of 0.02% deviation from the period was calculated. Therefore I simply made the periods a minimum of half a second so that the error was fairly minimal when basing the times off the last received packet thus accounting for offset. Applying this controlled viewing of the issue to when my program was analyzing packets, my program interpreted as a slowdown in the network because one of the packets too longer to arrive. The microseconds were taken into account but as you can see the drift directly impacted the timing. Even with including a deadzone and since the drift was essentially random within a certain percentage, the error was not deterministic and therefore was not truly periodic. It could be a function of the resolution or phase difference between what the get time function call uses, the real time clock, and the system clock. If viewed as a phase difference then this shows intervals like wavelengths where we are either leading or lagging the original clock. If viewed as an intelligent and calculated attempt to keep the task periodic, then it still induced a net drift away from the desired period. Thinking that the issue may actually lie in the get time of day function and how it uses the clock, I searched for another way to put the timestamp with the packet that was sent for later verification that the system was in fact performing a periodic operation so that we don't false trigger a notification of change in the network speed. I also looked into having the system itself encapsulate the timestamp into the packet. There is an option for the set socket options function to put a timestamp into the socket message that can be extracted later. It seemed that there were options for the software or the hardware to automatically attach a time stamp on either the receive or transmit directions. This seemed like an even more ideal implementation, but I didn't appear to have the resources to implement the reading of the timestamps. It also appeared to have much more overhead before and after the packet enters the network realm.

After I finally realized that I was not going to be able to use the kernel or the user space to ensure the periodic sending, I shifted my focus to resetting the router on total packet loss. This was the primary focus of the project. So I started by attempting to create a link between the client and the server which posed a new suite of issues. These issues were purely logical implementation of an application layer based connection using UDP. I wanted the client and the server to be started up in any order. Also I wanted the client to be able to find the server automatically since in most home networks the IP addresses would be assigned dynamically by the DHCP server in the router. By hardcoding the IP address this would make it difficult for a home user to set up the system. The primary issues with this setup involved broadcasting before anything was received, listening for an acknowledgement, then attempting to rebroadcast again if none was received. I learned that I cannot transmit after a binding operation if nothing has been received yet. I am not sure if I was missing the proper settings to achieve this, but nothing I found would fix this problem. The workaround I determined is that I have to broadcast before binding, then attempt bind and wait to receive. If no message is received and you want to rebroadcast the finding message, you must close the socket that was bound and wait for the operating system to release it. In my experiments it takes about 10 seconds to timeout the socket binding, then you can rebroadcast and try again. I also learned that two programs on the same system cannot bind to the same port. The reasoning seems to be that all packets that come in on a port that is bound should only go to the program that is bound to it. This ensures that the program that is bound to the port doesn't lose any data by allowing it to go to another program. This seems similar to a program taking the semaphore for a specific port and not giving it away until it is shut down. It appears that the request is instantly denied when a program requests a bind and another program has already bound to it. It appears that this is because the operating system has no way to know when or even if the program will release the bind. This proved to be a very interesting learning experience in how servers work.

Conclusion

There were many things I would still like to implement, especially synchronizing the periods so that the time between packets is known even though the initial packet will arrive is not known. I still think this is a very exciting concept and is not really explored in the network realm because it requires two machines or at least two network interfaces. Other packet types might be better for this method and using the bits to exchange information in codes rather than string would greatly decrease the overall size of the packet. Improving the aforementioned lookup table may be easily done and may end up looking similar to the NAT table system in routers to keep track of expected packets and keep a counter of how many times they are encountered. This system would be a relatively passive system to monitor the network status and relative speed without the need for the router to deviate from is regular data path routine. Only one packet has to go through the router to receive a new reading after the initial packet. As opposed to two packets for Ping which would actually take four packets to compare the speed before to the current speed. So for long term monitoring I would think that this would be an interesting method. Obviously I don't know if the method is sound or if it will actually result in any gains over previous methods since I was unable to test it or test it with any other monitoring methods. However, if the goal of this project was to learn and expose new concept and issues, then I would say that this project has succeeded its goal and I will continue to look into it.

# Works Cited

*socket(7)*. (2014, 2 21). Retrieved 5 9, 2014, from Linux Manual page: http://man7.org/linux/man-pages/man7/socket.7.html

Garcia-Navarro, T. (2006, 7 1). *Simple UDP Server*. Retrieved 5 9, 2014, from Linux Kernel Newbies: http://kernelnewbies.org/Simple\_UDP\_Server

Gawenda, R. (n.d.). *c - Sending small UDP packets from the Linux Kernel to LOOPBACK won't work reliable*. Retrieved 5 9, 2014, from stackoverflow: http://stackoverflow.com/questions/18339174/sending-small-udp-packets-from-the-linux-kernel-to-loopback-wont-work-reliable

NG. (2010, 12 14). *C - Set a timeout for reading stdin*. Retrieved 5 9, 2014, from Stackoverflow: http://stackoverflow.com/questions/3711830/set-a-timeout-for-reading-stdin