An Efficient Network Management and Power Saving Wake-ON-LAN

Pranjal Daga[a,*], D. P. Acharjya[b], Senthil J.[c], Pranam Daga[d]

[a]School of Computing Science and Engineering, VIT University, Vellore, India
[b]Faculty, School of Computing Science and Engineering, VIT University, Vellore, India
[c] Faculty, School of Computing Science and Engineering, VIT University, Vellore, India
[d]Purdue University, West Lafayette, Indiana, United States

Abstract. In distributed systems a computer generally process information of distributed application or provide service in distributed system. Therefore, computers connected in distributed system need to keep on all time. It leads to the concept of Wake-on-LAN (Local Area Network). However, keeping on during the idle period is the wastage of power. Therefore, it is essential to save the power while being efficient in network management. In this paper we propose an improved Wake-on-LAN device that incorporates the usage of basic Wake-on-LAN technology into a network management and power saving product.

Keywords: Wake-on-LAN, distributed system, magic packet, traffic routing, network management, header checksum.

1 Introduction

In the earlier decades, computer systems were highly centralized within a single room. But, the merging of computers and communication technology has brought a profound influence on the way computer systems are organized. As a result single computer serving system is replaced by a large number of separate but interconnected computers. This leads to computer network. Therefore, a collection of autonomous computers are interconnected by a single technology to exchange their information [6]. This helps common people in different applications such as business expansion, access to remote information, person-to-person communication etc. With these applications people are now interested in computer network and distributed systems. However, there is a considerable overlap between computer network and distributed system. A distributed system consists of multiple computers that communicate through a computer network [4]. The[1] main objective of a distributed system is to solve large computational problems. The basic purpose is to coordinate the use of shared resources or provide communication services to the users. Therefore, a computer needs

[*]Corresponding Author, Tel.: +918148255990
pranjaldaga@gmail.com

to keep on while being in the distributed system. But, it is observed that a computer in a distributed system may not be busy always and remain idle sometimes. Therefore, power consumption is to be minimized while it is idle. At the same time, a computer should have efficient network management while being busy. Hence, power consumption and efficient network management are two major issues in distributed systems. The purpose was to resolve a major issue inherent to all computer networks: operating cost. Modern computers require a substantial amount of energy to operate. On a per-system basis, especially in a home environment, computers are economical. However in larger operations such as business networks, in which anywhere from tens to hundreds to even thousands of computers are deployed, operating costs are significant. To combat excessive power consumption, idle computers can be allowed to fall asleep. But under typical network configurations cannot be allowed as it would compromise network performance. Under sleep mode, a computer's network presence can be lost and data connections cannot be made. Furthermore, allowing computers to fall asleep is impractical because a user must manually wake the system if its services are required again. Wake-on-LAN [1] is an attempt to minimize the operating cost. A. P. Chandrakasan and Brodersen [3] have discussed minimizing power consumption in digital CMOS circuits. Mark Stemm and Randy H. Katz [5] have discussed measuring and reducing energy consumption of network interfaces in handheld devices. Grant Ayers et al. [2] developed a lightweight, cost-effective device known as Whack-on-LAN capable of forcing a hardware reset over Ethernet. Emulab [7, 8] is a time- and space-shared network test bed with hundreds of publicly accessible nodes.

In this paper, we propose an improved Wake-on-LAN which minimizes the cost while being efficient in network management. The design is an intermediate network device that is placed between client computers and the network at large. Under normal usage, when its client is actively communicating across the network, the intermediate device allows traffic to flow through as required. However when a client is inactive and falls asleep, the device maintains the client's network presence and allows connections to continue to be made. In this sleep-mode operation, the device simultaneously caches data for its client while waking it using the standard Wake-on-LAN implementation. After its client wakes, the cached data is sent to the client, ensuring network operation.

The rest of the paper is organized as follows. Section 2 discusses the hardware and software specifications required for improved Wake-on-LAN. Detail design of the proposed device is presented in Section 3 followed by a conclusion in Section 4.

2 Specifications

In this section, we propose the abstract view of the proposed device as shown in Fig. 1. The proposed device is an intermediate network device placed between client and network. During active schedule, the proposed device allows traffic to flow through as required by the client whereas in sleep mode it caches data for its client while waking it using the standard Wake-on-LAN implementation.

Fig. 1. Abstract view of the proposed device

2.1 Software Specifications

The device is an intermediate network device that needs to be able to process network data, receive network packets, and forward network packets. To accomplish this, the software needs to be able to direct traffic to proper addresses. Furthermore, it also needs to be able to detect functional statuses (awake or asleep) of its client as well as to be able to broadcast a magic packet to wake a sleeping computer. Finally, it needs to be able to 2 simultaneously monitor traffic on both client facing and network facing ends. Therefore, operation, the software component of the device needs to handle the following:

- Packet capture, analysis, and injection
- Packet caching
- Traffic routing (via an IP table of addresses)
- Status detection
- Wake-on-LAN via Magic Packet broadcast
- Parallel operation

2.2 Hardware Specifications

One of the motivations for the design of the device is to save power. Therefore, its power consumption must be considerably lower than that of its clients. To accomplish this task, the main hardware chosen was the Beagleboard-XM, a single-board computer. A brief overview of the relevant specifications of the Beagleboard-XM is listed below:

- Advanced RISC Machine (ARM) Cortex A8 CPU operating at 1GHz
- 512MB of DDR Random Access Memory (RAM)
- 4 Universal Serial Bus (USB) ports
- Ability to run Linux
- Programmable General Purpose Input Output (GPIO)
- Peak power consumption of 2W

Because the ultimate goal of the design is to be able to operate in a “headless” plug and play manner, the device incorporates a series of 8 light emitting diode (LED) status indicators driven by the Beagleboard-XM s programmable GPIO.

3 Design Details

This section discusses the software design details such as state diagram of software when it recieves traffic and state diagram of the software when it sends traffic of the proposed device. The design of the proposed intermediate device meets the basic objective of Wake-on-LAN by maintaining client's network presence and allowing connections to continue in sleep mode. In this mode, the device simultaneously caches data for its client while waking it using the standard Wake-on-LAN implementation. Clear idea on these design issues is discussed in the following section.

3.1 Software Design

The software implementation needs to first capture incoming traffic. Once it captures a packet it needs to populate an IP table where it will keep the addresses which data can be routed to. Then it determines if the computer that the data is ultimately destined for is available. If the destination is not active, it will cache the traffic, wake the computer, and then forward the data over. If the destination is active, the software will immediately forward the traffic to the destination. This method is valid for both incoming and outgoing traffic. The state diagram of the device when it receives traffic is presented in the following Fig. 2 whereas in Fig. 3 we represent the state diagram of the device when it sends traffic.

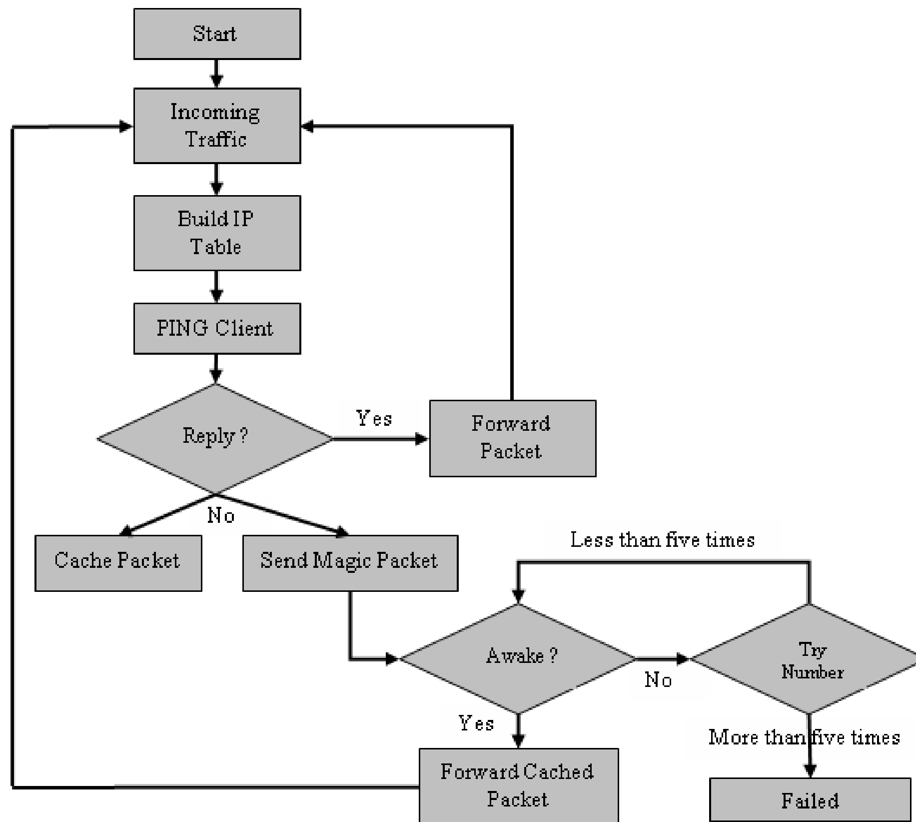

Fig. 2. State diagram of the device when it recieves traffic

- Packet Capture, Analysis, and Injection

Packet capture and packet injection is essential to the device's traffic forwarding function. When a client sends data to the network, it sends data to the intermediate device's client facing interface. The intermediate device must capture this packet and then inject it out its network-facing interface. Data sent to the client computer is handled in the same manner: the packets are captured from the network-facing interface and then injected to the client from the client-side interface. This capture and inject process is implemented using the packet capture (PCAP) library. We represent the general structure of a TCP packet in Fig. 4.

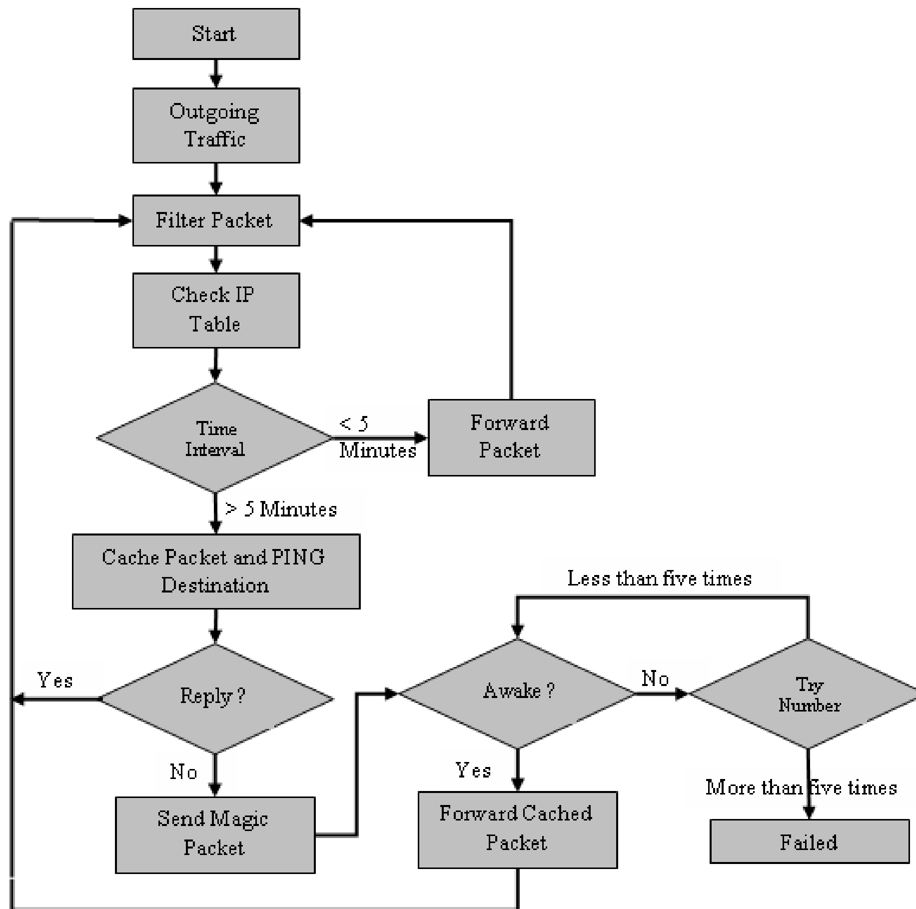

Fig. 3. State diagram of the device when it sends traffic

Fig. 4. The general structure of a TCP packet

When a packet is received, it is put into the network stack, in a receiving process. As the packet moves through the receiving process, it progresses up the network stack. At each stage of the process, a header is read and removed before going to the next step. This occurs starting at the Ethernet header, where the source and destination hardware MAC addresses are contained. After the Ethernet header is processed, the IP header, where source and destination IP addresses as well as the IP header checksum, is processed. Then, the transmission control protocol (TCP) header, where the pack-

et's flags and port information is contained, is processed. More header information is processed if they exist. Eventually, only the payload data remains. This payload data is then forwarded to the application that requested it.

In the case of the intermediate device, none of the payload data is relevant to any of its applications. All network traffic it receives is directed at either the client or the rest of the network. Thus, to the Ubuntu Linux operating system upon which the software implementation is built upon, all of the data moving through the network stack is garbage and will be discarded unless specifically preserved. This is where the PCAP library becomes useful.

The function `pcap_loop()` captures packets continuously from a specified Ethernet interface using a preset filter. In the case of the packet capture, this filter must be set to only accept data with destination to IP address of either one of the device's Ethernet interfaces. Upon capture of each packet, `pcap_loop()` calls a callback function from which analysis and processing can be run on the packet and the other functions, to control the LED status light, to cache packets, to build the IP table, to inject packets, and to broadcast the Magic Packet, are run. After a packet is captured, it is temporarily stored in serial form as a string of characters. Since a character is a single byte, data can be easily parsed through a method called typecasting. Typecasting allows pointers to structures types in C to be used frames. To locate specific data in the packet, such as the Ethernet header or the IP header, all one needs to do is create a structure and frame it over the desired region. To frame the Ethernet header, one would just need to create a pointer to a structure defining the contents of the Ethernet header and frame it over the first 14 bytes of the packet. To frame the IP header, one would simply do the same except with a pointer to an IP header structure and place it starting at the fifteenth byte of the packet. In pseudo-code it would look like the following:

```
string Packet;
struct Ethernet_header * eth_hdr;
eth_hdr = (struct * Ethernet_header *)Packet;
struct IP_header * ip_hdr;
ip_hdr = (struct * IP_header *)(Packet + 14);
```

The IP header needs to be modified in two ways, first the destination and source IP addresses need to be properly changed and secondly, the header checksum field must be filled with a recalculated checksum value. In the case of a packet sent from network to client, the destination address field originally contains the IP address of the Beagleboard-X's network-facing interface. This is because computers on the network only see the Beagleboard-XM and not the client computer that is shielded behind it. The Beagleboard-XM effectively acts as a secondary router to its client. Therefore, the destination IP address of the packet needs to be modified to reflect the IP address that the Beagleboard-XM's Linux operating system assigned to the client.

The IP addresses of the packets originating from the client computer do not need to be modified as they proper assignation is taken care of internally by Linux operating system. However, these packets must still be captured in the case that their destination computer is asleep and thus need to be cached by The Engine. Upon modification of any component of the IP header, a new IP header checksum needs to be calculated

and appended into the proper field. The IP header checksum is simply the ones' complement of the one's complement sum of the bytes in the IP header. This is an important step in IP header modification because if the checksum is detected to be erroneous by the destination computer, the entire packet is designated as corrupt and promptly discarded.

The checksums of possible headers following the IP header do not need to be modified as they pertain to the payload data as well as their internal values. These contents are never tampered with so no action needs to be done. For example, in the case of a TCP packet, the checksum is computed over the TCP header and its data. All of this higher-level data needs to be preserved; therefore, the checksum does not need to be recomputed.

- Packet Caching

If and when The Engine detects that the destination computer to which a packet is sent is asleep, the traffic is temporarily cached. To cache this data, the strings of characters in which the pcap_loop() function stores captured packets is stored into a global array. Packet data is cached into the global array until The Engine determines the destination computer is awake again. After confirmation of wake, each element of the array is modified and injected to the destination computer.

- Traffic Routing via IP Table

The IP Table is a dynamically allocated global memory structure that essentially routes outgoing traffic from the client. The IP Table is preset with a default connection timeout value. Whenever the client sends a packet to a machine, the IP Table is referenced to determine the time since last connection. If this time interval was greater than the preset timeout value then the program will cache the packet and send out a Magic Packet to the connection destination. However, if the time interval was less than the timeout value then the device assumes the destination is awake, and will forward the packet directly to the destination address. If the IP address had not been previously recorded, a new IP Table entry will be added and a timestamp will be recorded for future reference.

- Status Detection

Status detection of wake or sleep states of computers is implemented using a PING function. The device is set to ping the client every upon program execution but can also be set to ping at specific time interval. The PING function added a select function to include a default timeout to determine a lack of destination response. The default system call recvfrom() function does not support a timeout setting.

```
/* PING FUNCTION */
FD_SET(sockfd, &rfds);
tv.tv_sec = 1, tv.tv_usec = 0;
retval = select(sockfd+1, &rfds, NULL, NULL &tv);
```

With the code above, the PING is able to have a timeout value set by tv.tv structure. As an example, it is set as one second. Based on the response of the client from the PING function the device will decide to either forward the packet or to cache and send a magic packet.

- Wake-on-LAN via Magic Packet

The device sends out a magic packet via a user datagram protocol (UDP) broadcast containing the destination machine's MAC address. The magic packet is simply a packet with six 0xFF values followed by sixteen repetitions of the target's MAC address. The MAC address can be set either via user input from The Engine directly.

- Parallel Operation

Parallel operation is required in The Engine in order to wake a sleeping computer through the broadcast of a magic packet while simultaneously continuously caching incoming packets through a second instance of pcap_loop(). To run operations, also known as threads, in parallel, two functions of the PTHREAD library are required.

- `pthread_create()` – used to create a thread.
- `pthread_join()` – used to join threads.

Specifically, in the code, two threads of type pthread_t are created. One thread is to run the function to broadcast a magic packet. The second thread is to call a second instance of pcap_loop() to store packets into the global array. After the completion of either thread, they are joined together using the pthread_join() function and The Engine continues onto its functions. The code implemented in the Engine code is replicated below for clarity.

```
pthread_t thread1;
pthread_t thread2;
/* starts thread 1 to call Magic Packet function */
pthread_create(&thread1, NULL, &magic, NULL);
/* starts thread 2 to call pcap_loop() function to */
pthread_create(&thread2, NULL, &cachePackets, NULL);
/* join the two threads */
pthread_join(thread1, NULL);
pthread_join(thread2, NULL);
```

3.2 Hardware Specifications

The hardware design of this project created an LED status indicator for users. It displays the device's operating state in real-time, providing users with visual confirmation of device function or malfunction. Therefore, the user can manage and troubleshoot a network based on the status indicators. This is especially useful in management of large networks, where a status report of the sub-networks under the intermediate device is important. The LEDs also fulfill a debugging role as it indicates all of the statuses: capture, process, sleep status, and Wake-on-LAN. The hardware design is an add-on to the Beagleboard-XM. It consists of software and hardware elements.

The software design deals with the C code that implements Beagleboard-XM's general purpose input/output (GPIO) drivers. The hardware design pertains to the physical board design the LED indicators.

- Hardware for GPIO Drivers

The Beagleboard-XM comes with GPIO pins, to which the additional hardware board consisting of 8 LEDs is attached. This made the initial hardware design simple as all of the work is done internally on the Beagleboard-XM. No voltage regulator or jumpers were required as the device is already programmed to maintain a sufficient amount of voltage to drive any LEDs under 5V. Therefore, the final hardware add-on design for this device consisted of a circuit of 8 LEDs and 8 1kΩ in series with the GPIO connectors. These resistors ensured that the forward bias current that goes through the LEDs will not burn the LEDs. The most difficult aspect of the design was finding a suitable connector for the Beagleboard-XM's GPIO pin array. Once all the circuits are completed on the process control block (PCB) board, the software will take care of everything else.

4 Conclusion and Future Extension

The goal of the device is to ensure that packets are forwarded to the correct location on the network and the machines can be waken when required. All of the testing verification passed as stated in the design review. Throughout the research and development process, many publicly available tutorials were consulted to achieve a grasp of network programming. The performance of the device is tested not only in a closed environment, but also under real traffic scenarios and found satisfactory. Despite the overall success, there were several areas that were left unaddressed. Specifically, The Engine and underlying software contains memory leaks because it was a failure to allocate all of the memory that was assigned throughout the runtime. Furthermore, the end product was not a completely automated device. The intention of the project was to allow the device to work without any user configuration. Finally, the product packaging was very rudimentary and could have been created for a more durable product.

In future work, the software memory issues will be addressed. Though they did not affect performance in demonstration, these bugs need to be fixed before this product can be reliably used in real-world applications. Furthermore, it is planned to completely automate the device so it can be used as a plug-and-play product. Finally a case design could be created so the device is protected. In addition, improvements can still be made in code efficiency and product packaging.

The software libraries employed, namely PCAP, PTHREAD, and GPIO as well as their respective documentation are available in the public domain. The reliance on publicly available information creates little ethical concern as credit is given to the source parties.

5 References

1. AMD Corp. Magic Packet Technology. White Paper 20213, Rev A, AMD, Nov. 1995. http://-www.amd.com/us-en/assets/content type/white papers and tech docs/20213.pdf.
2. Ayers, G., Webb, Kirk., Hibler, M., and Lepreau, Jay: Whack-on-LAN: Inexpensive Remote PC Reset, (2005).
3. Chandrakasan, A. P. and Brodersen, R.: Minimizing Power Consumption in Digital CMOS Circuits, In Proceedings of the IEEE, Vol. 83 (4), pp. 498–523, (1995).
4. Sahani, G. J., Suthar, Keyur and Rao, Udai Pratap: A Proposed Power Saving Scheme for Distributed Systems by Powering On/Off Computers Remotely, International Journal of Advanced Research in Computer Science and Software Engineering, Vol. 3 (2), pp. 207-210, (2013).
5. Stemm, Mark and Katz, Randy H.: Measuring and Reducing Energy Consumption of Network Interfaces in Handheld Devices, In IEICE Transactions on Fundamentals of Electronics, Communications, and Computer Science, (1997).
6. Tanenbaum, A. S: Computer Networks, Prentice_Hall, Inc., Upper Saddle River, New Jersey, USA.
7. White, B., Lepreau, J. and Guruprasad, S.: Lowering the Barrier to Wireless and Mobile Experimentation, In Proceedings of HotNets-I, Princeton, New Jersey, (2002).
8. White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C. and Joglekar, A.: An Integrated Experimental Environment for Distributed Systems and Networks, In Proceedings of the Fifth Symposium on Operating Systems Design and Implementation, Boston, MA, pp. 255–270, (2002).