



IS YOUR PRODUCT OR  
BOOK JOLT-WORTHY?

Dr.Dobb's

Dr.Dobb's Portal  
The World of Software Development

## Green Telnet

Reducing energy consumption is of growing importance. Jeremy and Ken create a "green telnet" that lets clients transition to a low-power, sleep state.

By Jeremy Blackburn and Ken Christensen, [Dr. Dobb's Journal](#)

Oct 23, 2008

URL:<http://www.ddj.com/cpp/211600219>

*Jeremy is a student in the Department of Computer Science and Engineering at the University of South Florida. Ken is a Professor in the Department of Computer Science and Engineering at the University of South Florida.*

Reducing the energy consumption of IT equipment is of growing interest for operational, economic, and environmental reasons. The existing client/server model that uses TCP connections tightly couples application state to connection state. Thus, when a client machine transitions to a sleep state to reduce its energy footprint, the TCP connection drops and the application state is lost on the server. Because of this tight coupling, an idle client machine must remain fully awake to avoid losing its TCP connection to the server and any application state associated with that connection. That is, the client machine must remain in this energy inefficient state or lose any outstanding work (state) in the server. As a result, users often permanently disable power management in client machines.

To address this, we have created a "green telnet" that lets clients transition to a low-power, sleep state to save energy and not lose their telnet session (and state) in the server. We have modified telnetd to recognize when a client machine goes to sleep, gracefully handling the loss of the TCP connection, and then buffering data for the sleeping client. When the telnet client wakes up, a new TCP connection is established and the data buffered in the server is delivered to the client. Our new "gtelnetd" daemon is a modification of the Minix 3 telnetd daemon ([www.minix3.org](http://www.minix3.org)) and is as distribution agnostic as possible by relying on `fork()` and minimal IPC primitives. To enable the client machine to go to sleep without losing its session, we wedge three processes between the network connection and the telnetd application. These three processes coordinate data transfer, data queuing, and sleep/wake state transitions in a manner that is transparent to the telnet session. We also make some changes to the telnet client and introduce several sleep-related control messages.

### Making the Client/Server Model Green

To make the client/server model green, we must allow for the client to go to sleep and thus lose its TCP connection to the server, but still maintain application state in the server. The first step in this process is to define the protocol with which the client and server will communicate power state changes. We have defined three control messages through which the client and server communicate information regarding power state change:

- `gtWSLE`, which the client sends to the server to indicate that it wants to go to sleep. The payload for this control message is a time indicating how long the client intends to be asleep (0 indicates an unknown sleep period).
- `gtWWAK` is issued by the client when it wants to wake up. The payload for this message is a byte count used to recover from any data loss that might have occurred.
- `gtRECO` is sent by the server to let the client know which port number it should reconnect to for the next sleep/wake cycle.

Figure 1 shows the flow of control messages when a client goes to sleep (TCP ACKs are not shown). Upon receiving notification from the host power-management system, the client sends a `gtWSLE` message to the server. While the client is asleep the server uses an intermediate queue to buffer data destined for the client. When the client wakes back up, it reconnects to the server and sends a `gtWWAK` message. The server responds by sending the client the next reconnect port via a `gtRECO` message and any data necessary for synchronization. To delineate control messages from normal data messages, a 2-byte flag field is inserted into data packets sent between a green telnet client and server as in Figure 2.

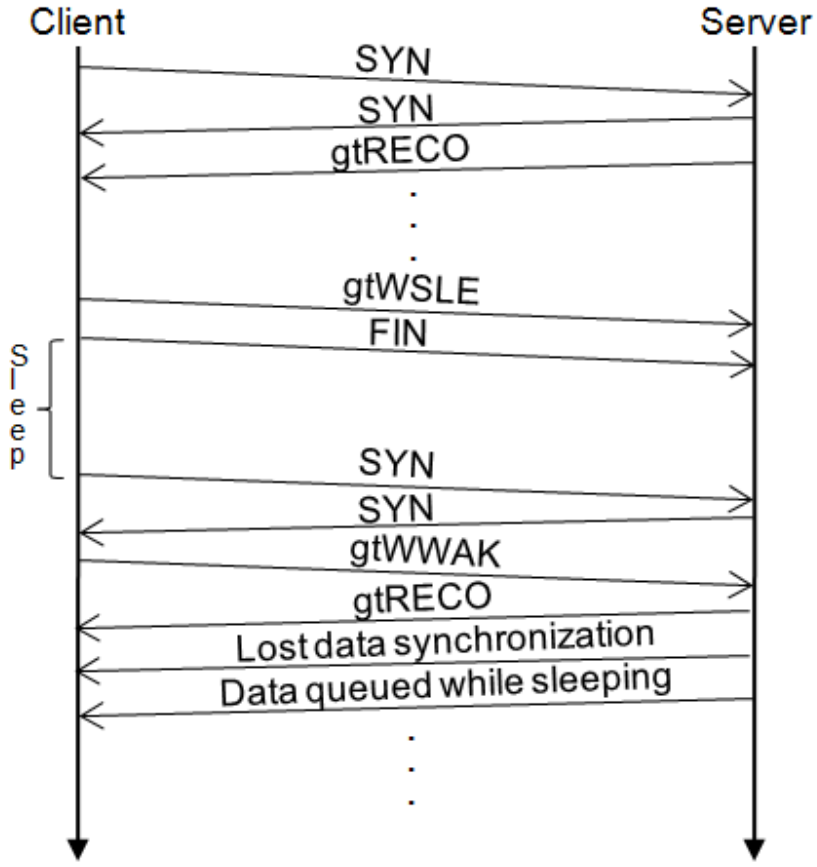


Figure 1: Power state change notification.

While the client sleeps, any running applications in the server continue to write data as normal; however, the data is not immediately sent to the client. Instead, we place the data in an intermediate queue and wait until the client reconnects. When the client transitions to a fully awake state and reestablishes a connection to the server (to its previously determined reconnect port), the server flushes any queued data back out to the client.

MAC	IP	TCP	gt	payload	CRC
-----	----	-----	----	---------	-----

gt flags = WWAK for want wake-up  
 WSLE for want sleep  
 RECO for reconnect  
 payload = gt command parameter if WWAK, WSLE,  
 or RECO, otherwise contains telnet data

Figure 2: gtelnet packet showing new gt field.

The reconnect, send data, and receive data processes compose the wedge we placed between the server application and the network connection. Figure 3 is an application-level block diagram of the reconnect process and its interactions with the send data and receive data processes. When the client is asleep, the reconnect process waits for a new connection from the client. When the client opens a new connection, we transition to "client is awake" state. The reconnect process accepts the new connection, sends the port number that the client should reconnect to for the next sleep/wake cycle, and spawns two processes—one to send data to the client and one to receive data from the client. The send and receive data processes execute in parallel, writing to and reading from the client independently. When the client transitions to a sleep state, the send and receive data processes cease executing and the reconnect process waits for the client to wake up and connect again. Because the server application itself never writes or reads directly to a socket, the sleep/awake transitions occur transparently, uncoupling the state of the application from the state of the

connection.

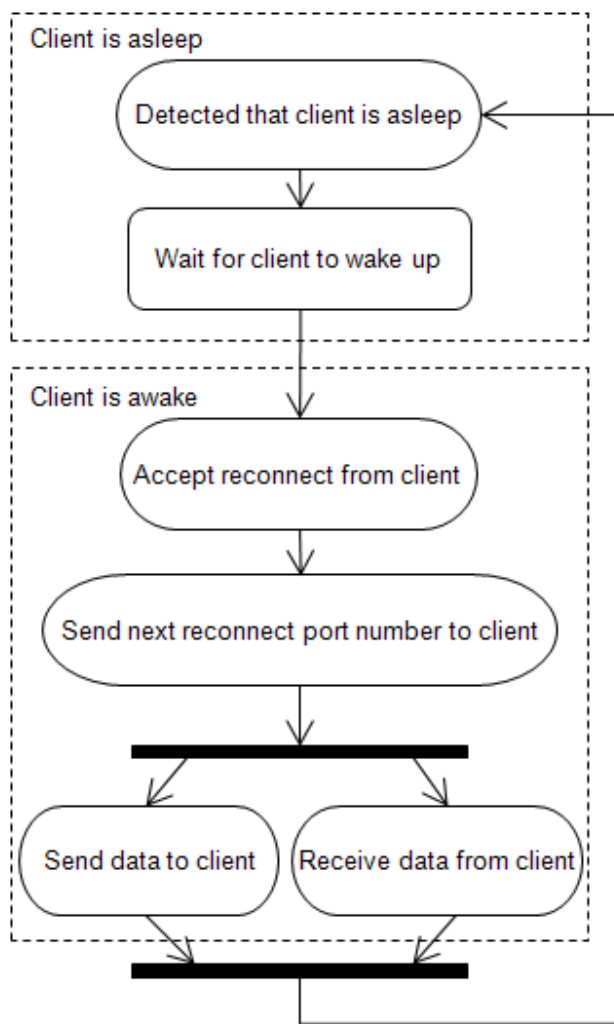


Figure 3: Server application-level process interaction.

## The `recv_dataA()` Function

When the reconnect process detects that the client has reestablished a connection, it spawns a process to receive the data from this connection. This new process calls the `recv_data()` function in Listing One. This function handles reading data from the network connection to the client and places it in an intermediate queue from which the `gtelnet` application reads. In this way, we have abstracted the underlying network connection uncoupling the state of the TCP connection from the state of the `gtelnet` daemon. The `recv_data()` function receives three variables as parameters. It requires the socket that the client is connected with (`s_client`), a FIFO buffer to write data that is read from the client (`recv_data_fd`), and the PID of the send data process to notify when the client goes to sleep (`send_data_pid`). Lines 13 through 15 read data from the client socket, `s_client`, and write the read data to `recv_data_fd`, which is the FIFO buffer that the server application reads data from. When the client is asleep and `recv_data()` is not invoked, the `recv_data_fd` has no new data placed into it, which appears as an idle connection to the server application. When a control message is detected from the client, we first determine whether or not it is a `gtWSLE` command, in which case, on line 23 or 42 we notify the send data process that the client has gone to sleep. At this point, the `recv_data()` function exits and data will not be read from the client until the client transitions back to an awake and operational state.

```

1. void recv_data(int s_client, int recv_data_fd, pid_t send_data_pid) {
2.     struct gserver_msg client_message;
3.     char gt_packet[BUF_SIZE];
4.     char * p;
5.     int bytes_read = 0;
6.     int i = 0;
7.     char last = 1;
8.     char *iacptr;
9.     char tb[1];

```

```
10. int temp_bytes_read = 0;
11.
12. while ((bytes_read = read(s_client, gt_packet, sizeof(gt_packet))) > 0) {
13.     iacptr = memchr(gt_packet, 255, bytes_read);
14.     if (!iacptr) {
15.         write(recv_data_fd, gt_packet, bytes_read);
16.     } else if (iacptr && iacptr > gt_packet) {
17.         if (iacptr == &gt_packet[BUF_SIZE - 1]) {
18.             write(recv_data_fd, gt_packet, bytes_read - 1);
19.             temp_bytes_read = read(s_client, tb, 1);
20.             if ((unsigned char)tb[0] == 255) {
21.                 bytes_read = read(s_client, &client_message, sizeof(client_message));
22.                 if (client_message.command == gtWSLE) {
23.                     (void)kill(send_data_pid, SIGALRM);
24.                     return;
25.                 }
26.             } else {
27.                 write(recv_data_fd, tb, temp_bytes_read);
28.             }
29.         }
30.     } else if (iacptr) {
31.         if (*(unsigned char *) (iacptr + 1) == 255) {
32.             if ((iacptr + 1 + sizeof(client_message)) <= &gt_packet[BUF_SIZE - 1]) {
33.                 p = (char *)&client_message;
34.                 iacptr += 2;
35.                 for (i = 0; i < sizeof(client_message); i++) {
36.                     *p = *iacptr;
37.                     p++;
38.                     iacptr++;
39.                 }
40.             }
41.             if (client_message.command == gtWSLE) {
42.                 (void)kill(send_data_pid, SIGALRM);
43.                 return;
44.             }
45.         }
46.     }
47. }
48. if (bytes_read < 1)
49.     exit(1);
40. return;
41. }
```

Listing One

## The `send_data()` Function

The `send_data()` function sends data over the wire to the client. Again, the server application itself never interacts directly with a socket, but rather writes data to an intermediate queue. When the client is awake, the `send_data()` function reads data from the intermediate queue and writes it to the client's socket. The `send_data()` function is in Listing Two. Lines 18 and 19 ensure that the client is still awake and awaits notification of the server application placing data in the queue. The condition on the loop breaks when the `client_wants_sleep` flag is set to True via the signal handler that catches a notification from `recv_data()` of the client transitioning to a sleep state. Next, starting on line 22 we loop through the intermediate queue, sending each message placed into the queue by the server application to the client. Lines 25 through 29 mark the replay buffer used to sync any data that had its transmission interrupted by the client going to sleep. Lines 30 through 44 simply close the queue if the client goes to sleep prior to transmission of all data in the queue, while lines 46 through 48 close the queue upon successful transmission of all its data.

```
1. void send_data(int s_client, int send_data_fd, struct replay_buffer *replay_buff) {
2.     struct gqnode dequeued_data;
3.     struct stat queue_stats;
4.     struct flock fl;
5.     char gt_packet[BUF_SIZE + 2];
6.     char data_notifier;
7.     char *p;
8.     char c;
9.     long queue_file_size;
10.    int temp = 0;
11.    int bytes_written = 0;
12.    int queue_fd;
13.    int status;
```

```
14.  int messages_written;
15.  int bytes_read = 0;
16.  int i = 0;
17.
18.  while (!client_wants_sleep &&
19.        read(send_data_fd, &data_notifier, sizeof(data_notifier)) > 0) {
20.    queue_fd = open (queue_file_path, O_RDWR);
21.    fl = acquire_queue_lock (queue_fd);
22.    while (read(queue_fd, &dequeued_data, sizeof(dequeued_data)) > 0) {
23.      if (!client_wants_sleep &&
24.          (bytes_written = write (s_client, dequeued_data.buf,
25.                                  dequeued_data.nbytes))
26.          replay_buff->byte_count = replay_buff->byte_count + bytes_written;
27.          for (i = 0; i < bytes_written; i++) {
28.            replay_buff->buffer[replay_buff->index] = dequeued_data.buf[i];
29.            replay_buff->index = (replay_buff->index + 1) % RB_MAX;
30.          }
31.          if (client_wants_sleep) {
32.            status = fstat (queue_fd, &queue_stats);
33.            queue_file_size = queue_stats.st_size;
34.            temp_queue_fd = open (queue_file_path, O_RDWR);
35.            messages_written = 0;
36.            while (read(queue_fd, &dequeued_data, sizeof(dequeued_data)) > 0) {
37.              write(temp_queue_fd, &dequeued_data, sizeof(dequeued_data));
38.              messages_written++;
39.            }
40.            close(temp_queue_fd);
41.            ftruncate(queue_fd, messages_written*sizeof(dequeued_data));
42.            release_queue_lock(queue_fd, fl);
43.            close(queue_fd);
44.            return;
45.          }
46.          ftruncate(queue_fd, 0);
47.          release_queue_lock(queue_fd, fl);
48.          close(queue_fd);
49.        }
50.    return;
51. }
```

Listing Two

## Changes to the Telnet Client

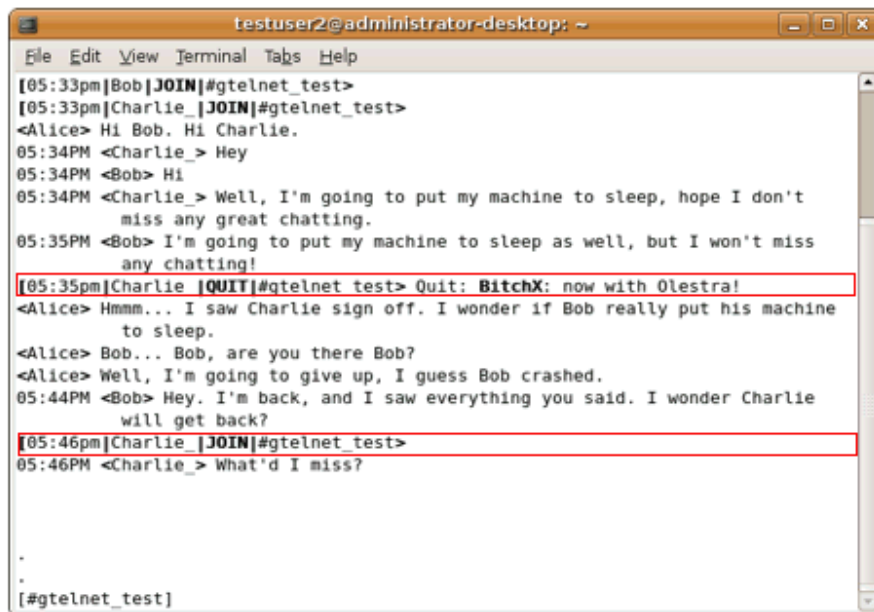
The telnet client has two major changes. Similar to the server, the client has a rolling byte count that keeps track of how many bytes of data it has received from the server. More importantly, however, the telnet client also has a thread that receives notification from the host operating system when the (client) machine changes power state. For the client to notify the server of changes in power state, the client must receive these changes from the host operating system. We have provided two possible implementations for this. The first implementation works on Ubuntu Linux and utilizes a script that is invoked by the host operating system's sleep and wake power management scripts. The script simply connects to each running client and passes the state change on. Our second implementation utilizes the power management API of Mac OS X. In this case, the client receives power state change notification directly from the operating system. Once the client has been made aware of a change in power state, it simply passes the notification up to the server with a *gtWSLE* or *gtWWAK* message.

When the client wakes up, it reconnects to the server and performs a two-step information exchange before normal communication resumes. As in Figure 1, the client first receives the next reconnect port number from the server. The client then sends the server its byte count, which the server uses to calculate any data discrepancies. The client then resumes normal operation. If the server determines that any data has to be resent to the client, the client receives it via normal communication channels. That is, while the data may have originally been sent by the server prior to the client going to sleep, if the client missed this data for any reason it is simply resent in the normal manner after the client wakes up.

## gtelnetd in Action

Figure 4 shows gtelnetd in action. Three clients are connected to an IRC server and chatting in the same room. Chatter "Alice" stays connected to the server the entire time, chatter "Bob" is using the gtelnet client, and chatter "Charlie" is using a regular telnet client. Both Bob and Charlie are logged into a shell account from which they run their respective IRC clients. At 05:33pm, Alice sees both Bob and Charlie enter the chat room. After some initial conversation, both Bob and Charlie send their machines to sleep at 05:35pm. Alice receives a message indicating that Charlie has disconnected from the IRC server due to a dropped TCP connection

to the server he is running his IRC client on. Bob, however, effectively never leaves the chat room due to his usage of the gtelnet client. When his machine goes to sleep, the gtelnetd server buffers all data that is transmitted. When Bob wakes-up his machine at 05:44pm, he sees the messages that Alice transmitted when his machine was asleep. Charlie, however, reconnects at 05:46pm and has not received any of the chatting that took place between 05:35pm and 05:46pm.



```
testuser2@administrator-desktop: ~
File Edit View Terminal Tabs Help
[05:33pm]Bob [JOIN]#gtelnet_test>
[05:33pm]Charlie [JOIN]#gtelnet_test>
<Alice> Hi Bob. Hi Charlie.
05:34PM <Charlie_> Hey
05:34PM <Bob> Hi
05:34PM <Charlie_> Well, I'm going to put my machine to sleep, hope I don't
miss any great chatting.
05:35PM <Bob> I'm going to put my machine to sleep as well, but I won't miss
any chatting!
[05:35pm]Charlie [QUIT]#gtelnet test> Quit: BitchX: now with Olestra!
<Alice> Hmm... I saw Charlie sign off. I wonder if Bob really put his machine
to sleep.
<Alice> Bob... Bob, are you there Bob?
<Alice> Well, I'm going to give up, I guess Bob crashed.
05:44PM <Bob> Hey. I'm back, and I saw everything you said. I wonder Charlie
will get back?
[05:46pm]Charlie [JOIN]#gtelnet test>
05:46PM <Charlie_> What'd I miss?
.
.
[#gtelnet_test]
```

Figure 4: gtelnetd in action.

## Next Steps: Spreading the Meme

The developed gtelnetd is a useable application. All source code is available at [sourceforge.net/projects/gtelnet](http://sourceforge.net/projects/gtelnet) and contributions are welcome. The techniques applied in the creation of gtelnet can be used in other client/server applications. Any interactive program in which data might be generated and delivery desirable, even though a user may not wish to respond in real time, is ripe for making power-state aware. While telnet is not used often anymore, SSH is popular and implementing power-state aware functionality should be very similar to our green telnet modifications. Instant messaging applications are another set of software in which clients may want to receive messages sent while the machine sleeps.

A Lawrence Berkeley National Laboratory technical report ([enduse.lbl.gov/Info/LBNL-45917b.pdf](http://enduse.lbl.gov/Info/LBNL-45917b.pdf)) stated that, "proper functioning of power management would achieve additional savings of 17 TWh/year." At \$0.10 per kWh, this is \$1.7 billion per year—a very large savings potential! Many, if not most, desktop computers have power management completely disabled due to the inconvenience it causes to users. By removing the application state's dependency on a network connection as we have done with gtelnetd, users will be less inclined to disable power management. This can result in large savings of money and also reduce the CO2 footprint of PCs.

## Acknowledgments

The development of this material is based on work supported by the National Science Foundation under grant CNS-0520081. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation (NSF). The authors thank Axel Vigo from the University of Puerto Rico, Mayaguez for his early prototypes of green client/server applications.

