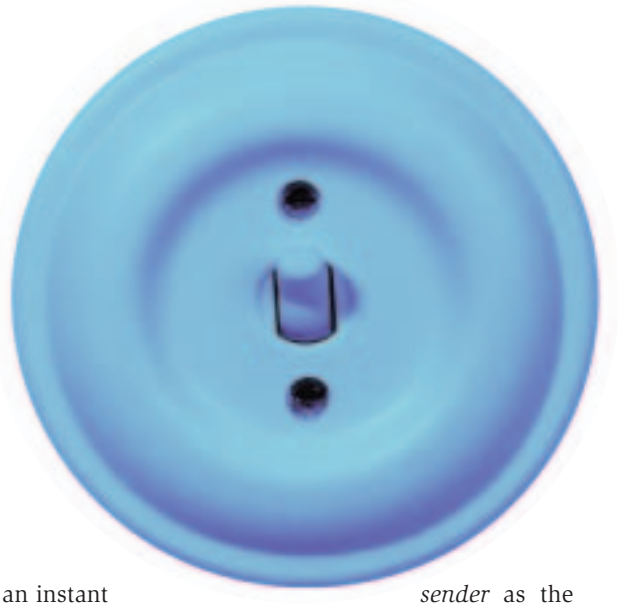
Remote control with a Jabber Bot

Inside



To get past a firewall and into a LAN, you need either a secret backdoor or a cooperative agent on the inside. A Jabber client on the LAN contacts a public Jabber server and wait for instructions trickling in as instant messages from its Internet buddies.

BY MICHAEL SCHILLI

Of course, one way to perform tasks on a local network from the Internet is to poke a hole through your firewall and connect to a local web server. Services like dyndns.org allow quasi-static access to the dynamic IP addresses that Internet providers assign.

An agent or "bot" (probably short for "Robot") makes life simpler: a messaging client on the inside of the firewall can attach to the public Jabber messaging network and accept commands in the form of text messages. The client I will describe in this article will only accept commands from clients on its buddy list, and it only supports four actions: load checking for the bot computer, querying the public router address (command: *ip*), and switching the lights on and off at my apartment in San Francisco (*lamp on|off*).

The *agent.pl* script (Listing 1) requires *Log::Log4perl* and logs transactions in a file called */tmp/agent.log*. Line 33 of the script creates a new *Net::Jabber::Client* object that implements an instant messaging client that will act as the agent.

Before *agent.pl* enters the main event loop in line 83, we need to define a few

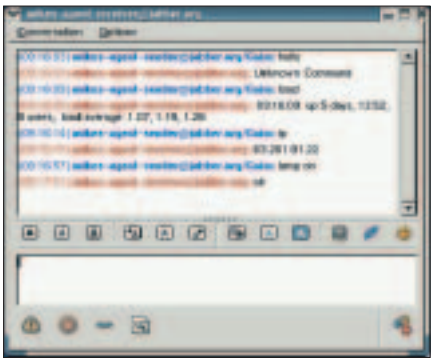


Figure 1: The "bot" behind the firewall runs commands sent to it by an Internet-based jabber client.

callbacks for several events in line 35ff. The *onauth* handler in line 68 will be called after the client has logged on to the server using the credentials specified in line 23. The handler then calls *RosterGet()* to fetch the buddy list and stores it in a global hash called *%ROSTER*. The *Presence()* method, which then follows, sends a *presence* message to all the clients on the roster to tell them that *agent.pl* is online. From this point onward, a *gaim* client with *mikes-agent-sender* as the logged on user will display *mikes-agent-receiver* as an active client on its buddy list (Figure 2).

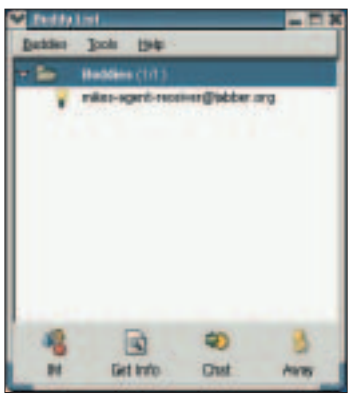


Figure 2: The "bot" now appears in the sender's buddy list.

The *message* callback in line 37 gets invoked when someone sends a message to *agent.pl*. Any client on the Jabber network could do this, and this is why line 45 checks if the sender is a friend. In our case, *mikes-agent-sender* is the only one allowed to send a message, as it is the only entry on the client's buddy list (see the "Installation" section). The function simply discards all other queries, logs an informational message and returns to the main loop in line 50.

The *getBody()* method in line 57 of the script extracts the control command sent with the message text and passes it to the *run_cmd* function defined in 93.

Execute() in line 83 connects to the Jabber server at *jabber.org* and logs in as *mikes-agent-receiver*. The main event loop recovers if the connection is temporarily lost, and should run indefinitely. If it does quit because too many errors have occurred, line 90 cleans up and quits the program.

To make sure the agent gets started when the Linux system is starting up, add the following:

```
x:3:respawn:su ➥
username -c➥
/usr/bin/agent
```

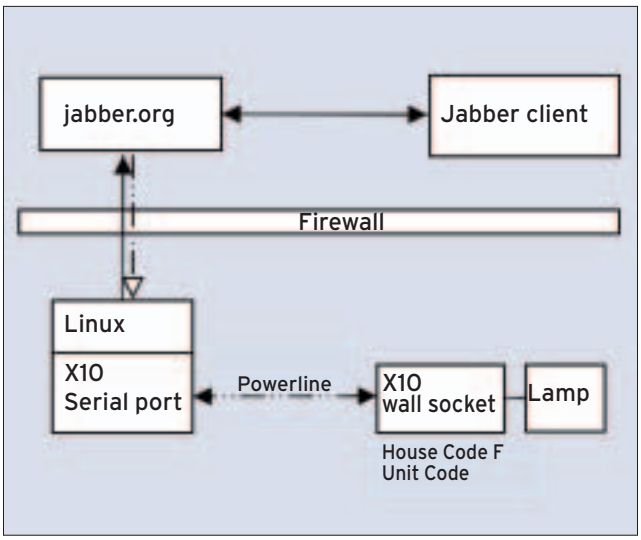


Figure 3: Overview of the Bot-controlled light switch.

to */etc/inittab*. This code also makes sure that the agent gets restarted immediately if it stops running for any reason.

Up Through the Down Pipe

The agent discovers my router's external IP address by sending a Web request to the public URL *http://perlmeister.com/cgi/whatsmyip*. The target is a simple script that returns the address of the requesting client:

```
print ➘
"Content-Type: text/html\n\n";
print $ENV{REMOTE_ADDR}, "\n";
```

agent.pl uses *LWP::Simple* to do this; the *get* function in line 99 gets the website content, if the text message the Jabber client receives is *ip*.

Determining current system load follows a similar pattern: line 107 calls *uptime* and passes the results back to line 55. The following call to *chomp* strips off a trailing newline and line 62 bundles the result into a message body; the *Send()* method sends this result off to the requesting chat partner.

But how does the agent running on a Linux computer go about switching on my bedroom lights?

Figure 3 shows the setup. In the USA, there is so-called X10 technology to transmit signals across the power cabling in our homes (of course it only works with the US type voltage), and communicate with computers via serial (or USB) interfaces. A wide range of X10 devices are available for various sorts of home automation. In addition to simple power switches, you will find X10-enabled surveillance camera systems, motion sensors, alarm systems, MP3 players, televisions, and metal detectors.

Each X10 control unit (Figure 4) has a house code (A-P) and a unit code (1-16) which the control unit (e.g. Figure 5) has to select in order to switch on the correct light (and not your neighbor's). X10 is not expensive: a four-component starter kit with all kinds of goodies and a remote control costs somewhere in the region of $50 to $100 at [3].

Listing *lamp.pl* shows a short script that sends codes out via the serial port to control the lamp. The script only uses *Device::ParallelPort* and *ControlX10::CM11* from CPAN, to address the unit using the house/unit code in line 38. A subsequent *send()* with the unit code and a "J" (for on) or a "K" (for off) pulls the trigger on the target device. The serial port used in this example is */dev/ttyS0* in line 34, since the little white box shown in Figure 6 is attached to the first serial port on my computer. Admittedly, my computer is not state-of-the-art, but that just goes to show how frugal Linux is with resources. Setting the baudrate to 4800 in line 35 makes sure the X10 device attached to the serial port reliably gets the message.

Limited Root Power

lamp.pl accesses the serial port on my computer and needs to run as *root* to do

Figure 4: The X10 control unit waits for signals and switches the power on or off.

Figure 5: The X10 control unit with a serial connector can send signals from the computer to the control unit over power cables.

Figure 6: Switching on a bedroom light via the Internet.

so. Line 29 checks the effective user ID and quits if the ID is not 0 for *root*. As the Jabber client runs with a non-privileged user ID, Listing *lamp.c* defines a C wrapper, which you can compile simply by calling

```
gcc -o lamp lamp.c
```

Setting the setuid via *chmod 4755 lamp* bit means that a 'normal' user can run the compiled binary *lamp*, and therefore the Perl script */usr/bin/lamp.pl* as *root*:

```
$ ls -l /usr/bin/lamp*
-rwsr-xr-x 1 root root 11548
 Oct  2 08:48 lamp
-rwxr-xr-x 1 root root   742
Oct  2 08:45 lamp.pl
```

In this configuration, only *root* can modify the *lamp.pl* script, but a normal user can run *lamp.pl* with the effective ID of *root*.

Bot Security

And now back to the bot:

To prevent any old Jabber client from sending commands, *agent.pl* only accepts messages from people in its ros-

Listing 1: agent.pl

```
001 #!/usr/bin/perl
002 ##############################
003 # agent -- Jabber bot
004 # operating behind firewall
005 # Mike Schilli, 2004
006 # (m@perlmeister.com)
007 ##############################
008 use warnings;
009 use strict;
010
011 use Net::Jabber qw(Client);
012 use Log::Log4perl qw(:easy);
013 use LWP::Simple;
014
015 Log::Log4perl->easy_init(
016   {
017     level => $DEBUG,
018     file  =>
019       '>>/tmp/agent.log'
020   }
021 );
022
023 my $JABBER_USER =
024   'mikes-agent-receiver';
025 my $JABBER_PASSWD = "*****";
026 my $JABBER_SERVER =
027   "jabber.org";
028 my $JABBER_PORT = 5222;
029
030 our %ROSTER;
031
032 my $c =
033   Net::Jabber::Client->new();
034
035 $c->SetCallBacks(
036
037   message => sub {
038     my $msg = $_[1];
039
040     DEBUG "Message '",
041       $msg->GetBody(),
042       "' from ",
043       $msg->GetFrom();
044
045     if ( !exists
046      $ROSTER{ $msg->GetFrom()
047             } ) {
048       INFO "Denied (not in "
049            "roster)";
050       return;
051     }
052
053     DEBUG "Running ",
054       $msg->GetBody();
055     my $rep =
056       run_cmd(
057         $msg->GetBody() );
058     chomp $rep;
059     DEBUG "Result: ", $rep;
060
061     $c->Send(
062       $msg->Reply(
063         body => $rep
064       )
065     );
066   },
067
068   onauth => sub {
069     DEBUG "Auth";
070     %ROSTER =
071       $c->RosterGet();
072     $c->PresenceSend();
073   },
074
075   presence => sub {
076       # Ignore all sub-
077       # scription requests
078   },
079 );
080
081 DEBUG "Connecting ...";
082
083 $c->Execute(
084   hostname => $JABBER_SERVER,
085   username => $JABBER_USER,
086   password => $JABBER_PASSWD,
087   resource => 'Script',
088 );
089
090 $c->Disconnect();
091
092 ##############################
093 sub run_cmd {
094 ##############################
095   my ($cmd) = @_;
096
097   # Find out external IP
098   if ( $cmd eq "ip" ) {
099     return LWP::Simple::get(
100     "http://perlmeister" .
101     ".com/cgi/whatsmyip"
102     );
103   }
104
105   # Print Load
106   if ( $cmd eq "load" ) {
107     return `/usr/bin/uptime`;
108   }
109
110 # Switch bedroom light on/off
111   if ( $cmd =~
112     /^lamp\s+(on|off)$/ )
113   {
114     my $rc =
115       system(
116       "/usr/bin/lamp $1");
117     return $rc == 0
118       ? "ok"
119       : "not ok ($rc)";
120   }
121
122   return "Unknown Command";
123 }
```

Figure 7: Using a gaim client to add the commanding agent to the list of approved senders.

ter. When a message arrives, line 45 checks if the sender is on the list, and refuses access if not.

The *presence* request handler defined in line 75 is empty and ignores any requests from clients wanting to put the agent on their buddy lists. *Net::Jabber::Client* comes with a default handler that accepts *presence* messages from just any other client on the network. This would not be so bad, but without further ado, it will put these clients on its buddy list. An empty *presence* handler stops this from happening.

Installation

When you install the bot, you have to set up its buddy list. The best way to do this is to use a *gaim* client ([4]), create two new jabber accounts *mikes-agent-receiver* and *mikes-agent-sender*, and have *mikes-agent-receiver* put *mikes-agent-sender* on its buddy list (Figure 7).

If both accounts are on line, the dialog shown in Figure 8 pops up on *mikes-agent-sender*, and you need to click "Authorize" to tell the server to allow the action to happen. *mikes-agent-sender* is then asked if it wants to put *mikes-agent-receiver* on its buddy list (Figure 9); of course, it makes sense to do this to enable the sender to click on the name in the buddy list to send a command to the bot.

After logging out, make sure the account *mikes-agent-receiver* is only used by *agent.pl* and not by other clients, to prevent them from messing up the buddy list, which provides the authorization mechanism.

When you launch *agent.pl*, *mikes-agent-receiver* should appear in the buddy list for *mikes-agent-sender* (Figure 2). The logfile, */tmp/agent.log*, logs the steps in case your setup needs some debugging.

Be careful when you are applying modifications; a tiny implementation error could tear a hole in your firewall – so watch out!

Of course, if you live in an area where the electrical infrastructure does not support X10, you'll need to find another way to communicate with your light switch. But, regardless of the project, these techniques will help you get started with building an instant messaging agent. ■

Figure 8: Adding the sender to the bot's buddy list.

Figure 9: Adding the bot to the sender's buddy list.

Listing 2: lamp.pl

```
01 #!/usr/bin/perl
02 ##################################
03 # lamp -- x10 light switch
04 # Mike Schilli, 2004
05 # (m@perlmeister.com)
06 ##################################
07
08 use warnings;
09 use strict;
10
11 use Device::SerialPort;
12 use ControlX10::CM11;
13
14 my $UNIT_CODE  = "F";
15 my $HOUSE_CODE = "1";
16
17 my %cmds = (
18   "on"  => "J",
19   "off" => "K",
20 );
21
22 die "usage: $0 [on|off]"
23   if @ARGV != 1
24   or $ARGV[0] !~
25       /^(on|off)$/;
26
27 my $onoff = $1;
28
29 die "You must be root"
30   if $> != 0;
31
32 my $serial =
33   Device::SerialPort->new(
34   '/dev/ttyS0', undef );
35 $serial->baudrate(4800);
36
37 # Adress unit
38 ControlX10::CM11::send(
39   $serial,
40   $UNIT_CODE . $HOUSE_CODE );
41
42 # Send command
43 ControlX10::CM11::send(
44   $serial,
45   $UNIT_CODE . $cmds{$onoff}
46 );
```

Listing 3: lamp.c

```
01 main(int argc, char **argv) {
02     execv("/usr/bin/lamp.pl",
03           argv);
04 }
```

INFO

[1] Listings for this article: *ftp://www.linux-magazine.com/Magazine/Downloads/50/Perl*

[2] "Jabber Developer's Handbook", Dana Moore and William Wright, Developer's Library, Sam's Publishing, 2004.

[3] X10 devices from: *http://x10.com*

[4] Gaim, the universal instant messaging client: *http://gaim.sourceforge.net*

THE AUTHOR

Michael Schilli works as a Software Developer at Yahoo!, Sunnyvale, California. He wrote "Perl Power" for Addison-Wesley and can be contacted at mschilli@perlmeister.com. His homepage is at http://perlmeister.com.