3-2009

Putting a Slug to Work

Konstantin Läufer
Loyola University Chicago, klaeufer@gmail.com

George K. Thiruvathukal
Loyola University Chicago, gkt@cs.luc.edu

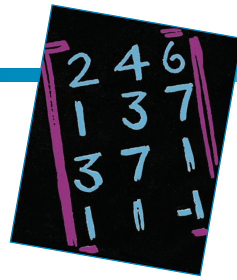
Ryohei Nishimura

Carlos Ramirez Martinez-Eiroa

Recommended Citation

Konstantin Läufer, George K. Thiruvathukal, Ryohei Nishimura, Carlos Ramírez Martínez-Eiroa, "Putting a Slug to Work," Computing in Science and Engineering, vol. 11, no. 2, pp. 62-68, Mar./Apr. 2009, doi:10.1109/MCSE.2009.35

Editors: Konstantin Läufer, laufer@cs.luc.edu
Konrad Hinsen, hinsen@cnrs-orleans.fr



PUTTING A SLUG TO WORK

By Konstantin Läufer, George K. Thiruvathukal, Ryohei Nishimura, and Carlos Ramírez Martínez-Eiroa

Although novel architectures such as cell processors, graphics processors, and FPGAs are growing in popularity, conventional microprocessor designs pack a punch in a small footprint and are widely supported by commodity operating system and development tools.

We've been busy during the past several months with an emerging multidisciplinary research project focused on environmental science in an urban setting. Given that Chicago is our hometown, matters of air and water quality are near and dear to our hearts (literally), so we hope to bring multiple people and organizations together to better understand them. Toward this goal, we've been exploring several technologies to support mobile and wireless distributed computing, which will play a crucial role in our project. In this installment of Scientific Programming, we're going to talk about our initial foray into embedded Linux running on the *slug*, which is the pet name for the Linksys NSLU2, a home "appliance" aimed at providing network-attached storage.

Although the slug is aimed at home users wanting network-attached storage (a form of storage not attached to any one computer but accessible from any computer on the LAN), we took immediate notice of this device as a prospective embedded host for other kinds of USB devices—in particular, those that can support various environmental sensing technologies. Its power footprint also attracted us: this completely fanless, thus silent, computer runs on a 12-V DC power input, which you can power with a cigarette lighter. If you're nonsmokers like us, you've probably been waiting for the day when you could put that cigarette lighter to use for something other than its original intended purpose. In any event, our hope is to use the slug as a host for environmental monitoring, which is becoming possible via a growing number of USB-capable sensors.

Although it has several physical challenges to address (which we won't cover here), this platform is only one of a whole class of computers that can run embedded Linux—and run it well, especially with all the work going into lower-power processors. The device firmware itself is Linux-based and uses the GNU toolchain and various utilities that most people already know and love. Because Linksys made the decision to use Linux, the firmware is bound by open source licenses, and its source code is freely available. Translation: hackers have already figured out how to replace the slug's firmware, so you can turn this device initially aimed at NAS into a full-fledged Linux environment and develop your own applications to run on it. (Please note that replacing the stock firmware voids your warranty.)

Even though Linksys has discontinued this product, you can still get your hands on one for roughly US$60 if you look around a bit. What we cover here also applies to various other similarly hackable devices, including network storage units and wireless routers. Before you buy, just make sure you do the research on how hackable the device is and what firmware supports which of the hardware features (for example, the USB ports in some wireless routers might not work).

Setting up the SlugOS

Let's now focus on the nuts and bolts. The first thing we did was to transplant the firmware on the device, starting with the extensive resources at www.nslu2-linux.org. Although we initially found the information perplexing, owing to the number of alternative distributions available, we ultimately settled on the SlugOS/BE distribution because it appeared to have the most momentum behind it, not to mention a large package database, which would eliminate the need to rebuild most of the packages we needed from scratch. Ordinarily, we wouldn't be bothered by having to build a few packages from scratch, but memory is severely constrained on the slug at 32 Mbytes RAM and 8 Mbytes flash (still better than your first PC, mind you), so we'd need to set up a cross-compiler to build a good number of the available free and open source software (FOSS) packages. Even so, the other advantage of SlugOS/BE is that most available packages have already been tested on the slug and, whenever possible, hand-tuned or written to keep the memory footprint low.

In the remaining discussion, we'll

talk about how to get a complete slug environment up and running. We'll also describe how to integrate the slug into your existing home or office computing setup, which will allow you to login remotely via SSH for your own experiments.

Step 1: Initial Setup

To set up the device, we recommend that you have an existing Linux setup somewhere on your LAN (your desktop or laptop will do):

- Install the NSLU2 flash utility appropriate for your host hardware (upslug2 for Linux and Mac OS).
- Download the latest SlugOS/BE binary image for NSLU2 from www.slug-firmware.net.
- Put the slug in upgrade mode following the instructions at www.nslu2-linux.org/wiki/HowTo/UseTheResetButtonToEnterUpgradeMode. Similar to many other home networking appliances (such as routers), you power off the device and then insert a pin or straightened paper clip into the pinhole near the back of the device. You then press and release the power button exactly as instructed.
- Flash the image following the instructions at www.nslu2-linux.org/wiki/SlugOS/UsingTheBinary. If you're on a Linux box (such as Ubuntu or a similar major distributions), you can just install the upslug2 utility via aptitude or apt-get.
- Wait for the slug to reboot.

Next, you'll have to install the operating system.

Step 2: Installing the OS

The next step requires your network to be set up properly—in particular, if this is your home network, we recommend that you set your private IP subnet to 192.168.1.0/255.255.255.0. By default, the slug has a fixed address, 192.168.1.77; you might also want to check in your router's Dynamic Host Configuration Protocol (DHCP) client table whether the slug has come up with a dynamic address. Another possibility is to connect the device to any computer that has more than one network port and make sure the additional port is configured to the same subnet. It's permitted to have more than one private subnet (just make sure they're not both 192.168.1.0). Now you just follow the instructions at www.nslu2-linux.org/wiki/OpenSlug/InitialisingOpenSlug:

- Log into the slug: `ssh root@192.168.1.77` (or actual dynamic address). The password is `opeNSLUg` (the letters in red must be capitalized).
- Initialize basic configuration: `turnup init`.
- Leave networking set to DHCP. We'll work on how to discover the hostname later, which eliminates the need for a static IP in most situations.
- Set the host name in accordance with your project naming scheme, such as luc-etl-slug0.
- Using the "vi" editor, remove the line containing `w_g_name` (domainname) from /etc/default/sysconf. It should now look like this:

```
[network]
hw_addr=
lan_interface=eth0
disk_server_name=luc-
   etl-slug1
bootproto=dhcp
```

- Preserve basic configuration in NVRAM: `turnup preserve`.
- Reboot.

Not bad, right? Next, we'll work on the discovery.

Step 3: Finding Your Device on the LAN

As indicated in Step 2, we're big fans of discovery, which is distributed systems speak for "I should be able to browse for the device when it's running." DHCP is vastly underrated: when you use it, you can actually discover all your attached devices (slug included) simply by going to the network router's administrative interface. (Most routers released within the past few years support this capability.)

In recent years, however, Zeroconf (www.zeroconf.org) has gained some popularity. This multicast framework for device announcement and discovery has its roots in AppleTalk and now lives as Bonjour (formerly Rendezvous). Zeroconf lets us browse the network of devices via a common name and look up various properties about them, notably their IP addresses. Translation: set up Zeroconf on your slug and on your computer (unless you're using a Mac, which means you already have it running), and you'll be able to find your slug by its common name as set in Step 2 (in our case, luc-etl-slug0). Ubuntu Linux also supports Zeroconf out of the box.

On the slug, you need to use the intrinsic packaging system (ipkg) to set up Zeroconf support via the Avahi project:

- First, make sure your package database is up to date: `ipkg update`.
- While you're at it, make sure all packages are current: `ipkg upgrade`.
- Install the Avahi daemon: `ipkg install avahi-daemon`.
- Make sure it's running: `/etc/init.d/avahi-daemon start`.

- (Optional) set up the Avahi utilities, which let you browse your full network from the device: `ipkg install avahi-utils`.

You'll also want to install Zeroconf if you're running on Windows or Linux. For Linux distributions, you should see packages named `avahi-*`. We use Ubuntu Linux, which provides two important packages: `apt-get install avahi-daemon avahi-utils`. Technically, you don't need to install `avahi-daemon`. We list it here just in case you're running an older version of (Ubuntu) Linux. As noted for the slug, `avahi-utils` lets you browse the network of devices via a command-line utility named `avahi-browse`. For Windows, you can download an installer from http://support.apple.com/downloads/Bonjour_for_Windows_1_0_5.

Once you install Zeroconf, you'll be able to access the device from any IP-enabled program (Web browser, SSH, and so on) via a common name such as <device-name>.local. We have several devices in our laboratory, so we use luc-etl-slugN.local (N = 1, 2, ...).

Whether you browse the network on the slug (*schneckle*, the South German diminutive for slug) or on your other computer (*feldberg*, in this case), you should now see the same list of services, as in Figure 1.

Step 4: Getting a Bigger Root Filesystem

Once you've completed Steps 1 through 3, you could consider yourself done in a technical sense, and you might well feel like calling it a day. You might even want to pour yourself a glass of wine to celebrate, but hold off until you make sure the following command works:

```
ssh root@<my-slug-hostname>.
local
```

Once you log in, though, you'll realize that this isn't your father's (or your mother's) Linux system. You can try various utilities to get an idea of the available resources:

```
top
cat /proc/cpuinfo
cat /proc/meminfo
df -h /
```

This might take you for a walk down memory lane to your first PC, the Commodore 64, the Timex-Sinclair SX80, or perhaps the Eniac or Z3. You'll be looking for every superlative or exaggeration until you realize that this is still a powerful computer. It has a reasonably zippy CPU and 32 Mbytes of RAM. You even have built-in flash storage; otherwise, how would your data persist?

But then you realize you want more.

This happened to us, and we decided that the best way to get "more" yet maintain the spirit of this device is to use an external USB stick to hold the root filesystem. After all, the slug has two USB ports (with the ability to support more than two via a USB hub, if you wish) and a network port, so we opted to pick up some USB sticks (a five-pack of 2-Gbyte sticks at the time of this writing was US$25). Having 2 Gbytes of storage makes our system even more upgradeable and usable for applications. In our work, we need to run a significant number of development tools and services, and we also use the device itself to host simple databases of environmental data until we can push it out to a remote server (such as a data warehouse), so additional storage is crucial to doing anything useful.

To get this benefit, insert your additional storage device into one of the slug's USB ports (doesn't matter which):

- The device might mount automatically in /media/<device-name>. You'll need to look for /dev/sda* or /dev/sdb* and unmount the partition. For example, if /dev/sda1 is mounted as /media/happy, just do `umount /media/happy`.
- The reason we need to ensure nothing is mounted is that we need to create a proper filesystem that will work nicely with Linux: ext3. In most cases, the device should be /dev/sda (you'll know if Linux tried to mount your FAT filesystem or whatever was on your USB drive in the first bullet). Try creating a partition on /dev/sda using fdisk (not covered here) and then do `mkfs.ext3 /dev/sda1`.
- If you're at all unsure what the device name is, it's usually harmless to just try /dev/sda. You can verify that you've got the right one by looking at the dmesg output and using grep to search for the detected device, `dmesg | grep sd[a-z]`.
- After creating the filesystem, we need to copy the existing root filesystem to the new device, `turnup memstick -i /dev/sda1 -t ext3`.

Once you've done these steps, you now have a usable root filesystem, but we still need to do some final steps to ensure that the USB stick is mounted as root, provided it's already inserted when you power up the slug. Go ahead and mount it on your Linux computer, say, on a temporary directory, /mnt:

- Add a disk label for your root filesystem, `tune2fs -L root /dev/sda1`.

```
avahi-browse -a -t
+ eth0 IPv4 EPSON Stylus C80 @ feldberg          _ipp._tcp            local
+ eth0 IPv4 Brother HL-2040 series @ feldberg    _ipp._tcp            local
+ eth0 IPv4 SqueezeCenter on feldberg            _http._tcp           local
+ eth0 IPv4 MythTV server on feldberg            _http._tcp           local
+ eth0 IPv4 SFTP File Transfer on feldberg       _sftp-ssh._tcp       local
+ eth0 IPv4 SFTP File Transfer on schneckle      _sftp-ssh._tcp       local
+ eth0 IPv4 feldberg                             _ssh._tcp            local
+ eth0 IPv4 schneckle                            _ssh._tcp            local
+ eth0 IPv4 feldberg [00:xx:xx:xx:xx:xx]         _workstation._tcp    local
+ eth0 IPv4 schneckle [00:xx:xx:xx:xx:xx]        _workstation._tcp    local
```

Figure 1. List of locally available services. The `avahi-browse` utility lets you browse your full network for available Zeroconf services.

- Add an entry to /mnt/etc/fstab, `LABEL=root / ext3 noatime 1 1`.
- Obtain the volume ID (its UUID), `vol_id /dev/sda1`.
- Reboot the slug without any USB drives present.
- Edit /linuxrc (using "vi" or your favorite editor) on the slug to refer to the UUID obtained earlier.
- Reboot the slug after inserting the USB stick.

We're almost there!

Step 5: Some Final Tweaking to Ensure Happiness

For the most part, you can consider yourself done, but this last step is essential if you plan to do serious development on your device—that is, you're not just planning on using it as an appliance. We're perfectionists, which means we can't live with annoyances, such as an incorrect time of day. Thanks to Step 4, which expanded the available storage for applications and data, this last step will let you grow the system as your needs and interests evolve:

- Turn off getty in /etc/inittab by commenting out the corresponding line using vi; getty isn't needed because there's no console attached. (You can probably add one using your other USB port, if you have a spare Zenith, VT, or Wyse dumb terminal lying around.)
- Populate the lists of available packages via the commands we covered in Step 3 (with `ipkg update` and `ipkg upgrade`).
- Set up the right time zone (optional but highly recommended), `ipkg install tzdata-right ; ln -s /usr/share/zoneinfo/right/CST6CDT /etc/localtime`.
- Set up automatic time synchronization, `ipkg install ntpclient`.
- Set up Optware following the instructions at www.nslu2-linux.org/wiki/Optware/Slugosbe.

Note that you now have two versions of ipkg, /usr/bin/ipkg to manage over 4,000 OpenEmbedded packages and /opt/bin/ipkg-opt to manage over 800 Optware packages. In particular, Optware has numerous useful server packages (media servers, print servers, Web servers, and so on). What gets a bit messy is the automatic starting and stopping of Optware services at boot and shutdown time. At www.nslu2-linux.org/wiki/OpenWrt, you'll see how to do this for another embedded Linux distribution called OpenWRT, but the instructions work equally well for SlugOS. Some packages exist in both places, and which version is better varies from package to package.

Optionally, if you want to be able to find your slug using DNS in addition to Zeroconf, set up Dynamic DNS using inadyn, a very lightweight DDNS client found in Optware: `ipkg-opt install inadyn`. You can set up /opt/etc/inadyn.conf according to the instructions at www.dyndns.org. If you want inadyn to start automatically, you must first create the startup script /opt/etc/init.d/inadyn with the contents shown in Figure 2, then create symbolic links to the script, and start it up:

- ln -s /opt/etc/init.d/inadyn /opt/etc/init.d/S60inadyn
- ln -s /opt/etc/init.d/inadyn /opt/etc/init.d/K60inadyn
- /opt/etc/init.d/inadyn start.

If ipkg ever starts failing, you might be out of memory. In that case, remove /var/lib/ipkg/* and /opt/lib/ipkg/*, then update the lists for each repository. In some cases, it helps to download the package file manually using wget and then installing it locally using ipkg. You might also want to add a swap partition to make software installation and configuration smoother. Because flash memory can handle only a limited number of writes, you should remove the swap partition once you put the slug into production. For the same reason, you should turn off most logging as well.

By the way, some older slugs are underclocked; if you're brave, see www.nslu2-linux.org/wiki/HowTo/OverClockTheSlug for a fix that requires removing a resistor from the PCB (at your own risk).

At this point, you should now consider enjoying your second glass of wine (if you haven't finished the bottle

```
#!/bin/bash

CONF=/opt/etc/inadyn.conf
PROGRAM=/opt/bin/inadyn

if [ ! -f $CONF ] ; then
  echo "No configuration file, exiting"
  exit 2
fi

# See how we were called.
case "$1" in
 start)
  start-stop-daemon -S -x $PROGRAM -- \
   --input_file $CONF --background
  ;;
 stop)
  start-stop-daemon -K -x $PROGRAM
  ;;
 *)
  echo "Usage: inadyn {start|stop}"
  exit 1
esac

exit 0
```

Figure 2. Startup script. By creating this script, you're well on the way to automatically starting inadyn.

already) because you've really earned it. You now have a complete environment to try some of our examples and sample applications that we developed in our group.

Applications

The next question is how to put the slug to use. In general, the slug excels at providing services that require relatively little CPU power and should be always available without drawing a lot of current or requiring a full-fledged server. Let's look at some of the wide-ranging possibilities that you can combine freely as long as you have memory left or add a swap partition on a conventional hard drive.

Media Server

Using the slug as a media server probably comes closest to what it was marketed for originally. You can plug an external USB hard drive into the second USB port, or you can use a USB hub to connect additional drives or other devices.

Depending on your specific needs, you'll want a combination of these services:

- Network File System (NFS),
- CIFS (Samba),
- Firefly (mt-daapd) media server for iTunes,
- UPnP media server, and
- Podget or some other automated podcast downloader.

Just keep in mind that the slug doesn't have the capacity to perform any CPU-intensive tasks such as media transcoding.

Print Server

Unless your printer is already network-enabled, you need to connect it to a specific computer's USB port (via a suitable adapter in some cases). This isn't convenient if, say, you have only laptops in your household.

Commercially available print servers solve this problem, but the slug handles it equally well along with lots of other functions. None of these embedded servers, slug included, have the power to run a full-fledged installation of the Common Unix Printing System (CUPS) that provides spooling and transformation of documents into PostScript or other printer languages. Instead, they expose the printer directly using the AppSocket protocol and assume that all the work happens on the client.

To set up your slug as a print server, all you need is p910nd from Optware. You'll have to run one instance of p910nd for each printer you're exposing. Unlike the print servers embedded in typical network printers, using a slug in this way lets you choose a nice host name. In addition, you can expose each printer as a Zeroconf service so your clients find them easily.

Telephony Server

Believe it or not, most of the setup described in a previous issue[1] runs fine on a slug. Most of the work the Asterisk telephony server does in this minimal setup is routing Session Initiation Protocol (SIP) packages, which play a role in setting up a call, while the actual voice traffic occurs between the call's resulting endpoints. The key is to avoid CPU-intensive codecs or other features that could bog down Asterisk. Consequently, CPU utilization remains below 10 percent even when a call is active, and the number of calls is usually very low for a home setup.

User-Space IP Address Registration Server

As part of our research project into

distributed sensor networks, we needed a service for keeping track of the IP addresses of nodes as they become available or unavailable. Although (Dynamic) DNS serves a similar purpose, we wanted something lightweight that we could run in user-space (not requiring privileged access) and evolve as needed.

In our network, we want to be able to create namespaces and register individual nodes within them. We take a resource-oriented approach typically associated with the representational state transfer (REST) architectural style, meaning that each thing that matters is an addressable resource that has at least one uniform resource identifier (URI) and supports a standard set of operations exposed as HTTP methods, including PUT (create), GET (retrieve), POST (update), and DELETE (delete).

The registry and its namespaces and nodes map naturally to this style:

- The registry lives at http://host/registry.
- To create a namespace, we submit a PUT request to http://host/registry, where the payload is a representation of the namespace to be created—that is, the namespace's name, such as myns.org. Once we create this namespace, it has the URI http://host/registry/myns.org.
- To delete a namespace, we submit a DELETE request to that URI.
- To retrieve a namespace, we submit a GET request to the same URI. Now, following the resource-oriented approach, we get back a suitable representation of the resource, in this case, a list of nodes currently registered within this namespace.
- Namespaces don't support renaming, which we would otherwise have mapped to the POST method.

Similarly, to register a node within a namespace, we submit a PUT request to the namespace's URI, http://host/registry/myns.org, where the payload is a representation of the host registration. In practice, this means a Web form with these parameters:

- host=myhostname,
- ip=10.20.30.40, and
- ttl=3600 # time-to-live in seconds.

The resulting URI for the registered host is http://host/registry/myns.org/myhostname. To retrieve a registered host, we submit a GET request to this URI and get back a representation like the one we submitted to create the host. To delete a registered host, we simply submit a DELETE request to the same URI, and to update a registered host's registration information, we submit a POST request to the URI whose payload is the updated registration information (such as the new IP address).

As part of his undergraduate research project in our group, Ryohei implemented this service on the slug. He chose Python as the implementation language because Python 2.5 (the one from Optware, not OpenEmbedded) runs out of the box on SlugOS/BE, has bindings to the lightweight SQLite database, and provides the BaseHTTPServer package as a good starting point for Web service development (the HTTP methods map to Python functions by name, such as do_PUT).

Because we use standard HTTP request methods, we don't even need to write a dedicated client. Instead, we simply use the cURL command-line client, which supports all request methods, form submission, file upload, and so on. In real life, the hosts would use the cURL client to register themselves, as you can see in Figure 3.

Webcam Server

In the broader context of our sensor network project, Carlos set up a slug as a webcam server for a class project. Besides integrating the required hardware drivers and software, he used PHP, XHTML, and JavaScript to develop a Web interface for the w3camd application. Consequently, a user can take pictures remotely and view them on the Web, and the application ensures that the drivers are loaded and the connected camera model is supported. Detailed project documentation is available at http://code.google.com/p/slurchin/.

At this point, you might be wondering whether you can run Java on the slug. The answer is a cautious, "yes, but not as well as Python."

We tried JamVM with GNU Classpath, intended as a replacement for Sun's Java SE API. Both packages are available in Optware and install without problems. Unfortunately, we couldn't get JamVM to run anything much beyond a "Hello Slug" console app. Some developers have reported success with older versions of JamVM, but we haven't had a chance to cross-compile it yet for the slug. We'll also try to cross-compile GCC with support for the GNU Java compiler (gjc) in our future work.

On the upside, the phoneME implementation of the Java ME platform does work. But because it targets mobile phones and other limited devices, it comprises a stripped-down API that's missing some of the packages servlet containers such as Jetty expect. Fortunately, an embedded subset of Jetty also works. The MinimalServlets example represents this subset and serves as our starting point for further exploration on the Java ME side.

```
curl http://schneckle.local/registry
# empty response: no namespaces in registry
curl -X PUT -d myns.org http://schneckle.local/registry
curl -X PUT -d myotherns.org http://schneckle.local/registry
curl http://schneckle.local/registry
http://schneckle.local/registry/myns.org
http://schneckle.local/registry/myotherns.org
curl http://schneckle.local/registry/myns.org
# empty response: no hosts in namespace
curl -X PUT -d host=myhost1 -d ip=10.20.30.40 -d ttl=3600 \
 http://schneckle.local/registry/myns.org
curl -X PUT -d host=myhost2 -d ip=11.22.33.44 -d ttl=3600 \
 http://schneckle.local/registry/myns.org
curl http://schneckle.local/registry/myns.org
http://schneckle.local/registry/myns.org/myhost1
http://schneckle.local/registry/myns.org/myhost2
curl http://schneckle.local/registry/myns.org/myhost1
namespace=myns.org
host=myhost1
ip=10.20.30.40
ttl=3600
curl -X POST -d ip=17.27.37.47 -d ttl=7200 \
 http://schneckle.local/registry/myns.org/myhost1
curl http://schneckle.local/registry/myns.org/myhost1
namespace=myns.org
host=myhost1
ip=17.27.37.47
ttl=7200
curl -X DELETE http://schneckle.local/registry/myns.org
curl http://schneckle.local/registry/myns.org
HTTP/1.0 404 Not Found
curl http://schneckle.local/registry
http://schneckle.local/registry/myotherns.org
```

Figure 3. Registry client code using cURL. Because we use standard HTTP request methods, we don't even need to write a dedicated client. Instead, we simply use the cURL command-line client, which supports all request methods, form submission, file upload, and so on. In real life, the hosts would use the cURL client to register themselves.

In any event, please stay tuned. We hope to cover progress on this topic in the near future.

Acknowledgments

George and Konstantin are grateful to Chandra Sekharan, chairperson of Loyola University Chicago's Department of Computer Science, for having funded our initial equipment purchases. Carlos is grateful to Corby Schmitz for general guidance and security-related advice on his webcam project.

Reference

1. G.K. Thiruvathukal and K. Läufer, "What I Did on My Summer Vacation," *Computing in Science & Eng.*, vol. 10, no. 6, 2008, pp. 76–81.

Konstantin Läufer is a professor of computer science at Loyola University Chicago. His research interests include programming languages, software architecture and frameworks, distributed systems, mobile and embedded computing, human-computer interaction, and educational technology. Läufer has a PhD in computer science from the Courant Institute at New York University. Contact him via www.cs.luc.edu/laufer.

George K. Thiruvathukal is an associate professor of computer science at Loyola University Chicago and is now an associate editor in chief of this magazine. His technical interests include parallel/distributed systems, programming language design/implementation, and computer science across the disciplines. Thiruvathukal has a PhD in computer science from the Illinois Institute of Technology. Contact him via http://gkt.etl.luc.edu.

Ryohei Nishimura recently graduated from Loyola University Chicago with a BS in computer science. His technical interests include programming languages, embedded systems, and distributed systems. Contact him at nishimura.ryohei@gmail.com.

Carlos Ramírez Martínez-Eiroa is a graduate assistant at Loyola University Chicago. His technical interests include Web development, distributed systems, and embedded systems. Ramírez Martínez-Eiroa has an MS in computer science from Loyola University Chicago. Contact him at crme1980@gmail.com.