Learn Linux, 101: Streams, pipes, and redirects

Getting comfortable with Linux plumbing

Ian Shields
Senior Programmer
IBM

14 October 2009

If you think streams and pipes make a Linux® expert sound like a plumber, here's your chance to learn about them and how to redirect and split them. You even learn how to turn a stream into command arguments. You can use this material in this article to study for the LPI® 101 exam for Linux system administrator certification, or just to learn for fun.

View more content in this series

Overview

This article grounds you in the basic Linux techniques for redirecting standard IO streams. Learn to:

- Redirect the standard IO streams: standard input, standard output, and standard error
- Pipe output from one command to the input of another
- Send output to both stdout and a file
- Use command output as arguments to another command

This article helps you prepare for Objective 103.4 in Topic 103 of the Linux Professional Institute's Junior Level Administration (LPIC-1) exam 101. The objective has a weight of 4.

Setting up the examples

Develop skills on this topic

This content is part of a progressive knowledge path for advancing your skills. See Basics of Linux system administration: Working at the console

In this article, we will practice the commands using some of the files created in the article "Learn Linux 101: Text streams and filters." But if you haven't read that article, or if you didn't save the files you worked with, no problem! Let's start by creating a new subdirectory in your home directory

called lpi103-4 and creating the necessary files in it. Do so by opening a text window with your home directory as your current directory. Then copy the contents of Listing 1 into the window to run the commands that will create the lpi103-4 subdirectory and the files you will use.

Listing 1. Creating the example files

```
mkdir -p lpi103-4 && cd lpi103-4 && {
echo -e "1 apple\n2 pear\n3 banana" > text1
echo -e "9\tplum\n3\tbanana\n10\tapple" > text2
echo "This is a sentence. " !#:* !#:1->text3
split -l 2 text1
split -b 17 text2 y; }
```

Your window should look similar to Listing 2, and your current directory should now be the newly created lpi103-4 directory.

Listing 2. Creating the example files - output

```
[ian@echidna ~]$ mkdir -p lpi103-4 && cd lpi103-4 && {
> echo -e "1 apple\n2 pear\n3 banana" > text1
> echo -e "9\tplum\n3\tbanana\n10\tapple" > text2
> echo "This is a sentence. " !#:* !#:1->text3echo "This is a sentence. " "This is a sentence. " "This is a
 sentence. ">text3
> split -l 2 text1
> split -b 17 text2 y; }
[ian@echidna lpi103-4]$
```

Redirecting standard IO

About this series

This series of articles helps you learn Linux system administration tasks. You can also use the material in these articles to prepare for Linux Professional Institute Certification level 1 (LPIC-1) exams.

See our series roadmap for a description of and link to each article in this series. The roadmap is in progress and reflects the latest (April 2009) objectives for the LPIC-1 exams: as we complete articles, we add them to the roadmap. In the meantime, though, you can find earlier versions of similar material, supporting previous LPIC-1 objectives prior to April 2009, in our LPI certification exam prep tutorials.

Prerequisites

To get the most from the articles in this series, you should have a basic knowledge of Linux and a working Linux system on which you can practice the commands covered in this article. Sometimes different versions of a program will format output differently, so your results may not always look exactly like the listings and figures shown here.

A Linux shell, such as Bash, receives input and sends output as sequences or *streams* of characters. Each character is independent of the one before it and the one after it. The characters are not organized into structured records or fixed-size blocks. Streams are accessed using file IO techniques, whether or not the actual stream of characters comes from or goes to a file, a keyboard, a window on a display, or some other IO device. Linux shells use three standard I/O streams, each of which is associated with a well-known file descriptor:

1. *stdout* is the *standard output stream*, which displays output from commands. It has file descriptor 1.
2. *stderr* is the *standard error stream*, which displays error output from commands. It has file descriptor 2.
3. *stdin* is the *standard input stream*, which provides input to commands. It has file descriptor 0.

Input streams provide input to programs, usually from terminal keystrokes. Output streams print text characters, usually to the terminal. The terminal was originally an ASCII typewriter or display terminal, but is now more often a text window on a graphical desktop.

If you already studied the article "Learn Linux 101: Text streams and filters," some of the material in this article will be familiar to you.

Redirecting output

There are two ways to redirect output to a file:

n>
: redirects output from file descriptor *n* to a file. You must have write authority to the file. If the file does not exist, it is created. If it does exist, the existing contents are usually lost without any warning.

n>>
: also redirects output from file descriptor *n* to a file. Again, you must have write authority to the file. If the file does not exist, it is created. If it does exist, the output is appended to the existing file.

The *n* in n> or n>> refers to the *file descriptor*. If it omitted, then standard output is assumed. Listing 3 illustrates using redirection to separate the standard output and standard error from the `ls` command using files we created earlier in our lpi103-4 directory. We also illustrate appending output to existing files.

Listing 3. Output redirection

```
[ian@echidna lpi103-4]$ ls x* z*
ls: cannot access z*: No such file or directory
xaa  xab
[ian@echidna lpi103-4]$ ls x* z* >stdout.txt 2>stderr.txt
[ian@echidna lpi103-4]$ ls w* y*
ls: cannot access w*: No such file or directory
yaa  yab
[ian@echidna lpi103-4]$ ls w* y* >>stdout.txt 2>>stderr.txt
[ian@echidna lpi103-4]$ cat stdout.txt
xaa
xab
yaa
yab
[ian@echidna lpi103-4]$ cat stderr.txt
ls: cannot access z*: No such file or directory
ls: cannot access w*: No such file or directory
```

We said that output redirection using n> usually overwrites existing files. You can control this with the `noclobber` option of the `set` builtin. If it has been set, you can override it using n>| as shown in Listing 4.

Listing 4. Output redirection with noclobber

```
[ian@echidna lpi103-4]$ set -o noclobber
[ian@echidna lpi103-4]$ ls x* z* >stdout.txt 2>stderr.txt
-bash: stdout.txt: cannot overwrite existing file
[ian@echidna lpi103-4]$ ls x* z* >|stdout.txt 2>|stderr.txt
[ian@echidna lpi103-4]$ cat stdout.txt
xaa
xab
[ian@echidna lpi103-4]$ cat stderr.txt
ls: cannot access z*: No such file or directory
[ian@echidna lpi103-4]$ set +o noclobber #restore original noclobber setting
```

Sometimes you may want to redirect both standard output and standard error into a file. This is often done for automated processes or background jobs so that you can review the output later. Use &> or &>> to redirect both standard output and standard error to the same place. Another way of doing this is to redirect file descriptor *n* and then redirect file descriptor *m* to the same place using the construct m>&n or m>>&n. The order in which outputs are redirected is important. For example,

`command 2>&1 >output.txt`

is not the same as

`command >output.txt 2>&1`

In the first case, stderr is redirected to the current location of stdout and then stdout is redirected to output.txt, but this second redirection affects only stdout, not stderr. In the second case, stderr is redirected to the current location of stdout and that is output.txt. We illustrate these redirections in Listing 5. Notice in the last command that standard output was redirected after standard error, so the standard error output still goes to the terminal window.

Listing 5. Redirecting two streams to one file

```
[ian@echidna lpi103-4]$ ls x* z* &>output.txt
[ian@echidna lpi103-4]$ cat output.txt
ls: cannot access z*: No such file or directory
xaa
xab
[ian@echidna lpi103-4]$ ls x* z* >output.txt 2>&1
[ian@echidna lpi103-4]$ cat output.txt
ls: cannot access z*: No such file or directory
xaa
xab
[ian@echidna lpi103-4]$ ls x* z* 2>&1 >output.txt # stderr does not go to output.txt
ls: cannot access z*: No such file or directory
[ian@echidna lpi103-4]$ cat output.txt
xaa
xab
```

At other times you may want to ignore either standard output or standard error entirely. To do this, redirect the appropriate stream to the empty file, /dev/null. Listing 6 shows how to ignore error output from the `ls` command, and also use the `cat` command to show you that /dev/null is, indeed, empty.

Listing 6. Ignoring output using /dev/null

```
[ian@echidna lpi103-4]$ ls x* z* 2>/dev/null
xaa  xab
[ian@echidna lpi103-4]$ cat /dev/null
```

Redirecting input

Just as we can redirect the stdout and stderr streams, so too can we redirect stdin from a file, using the < operator. If you already studied the article "Learn Linux 101: Text streams and filters," you may recall, in our discussion of sort and uniq that we used the `tr` command to replace the spaces in our text1 file with tabs. In that example we used the output from the `cat` command to create standard input for the `tr` command. Instead of needlessly calling `cat`, we can now use input redirection to translate the spaces to tabs, as shown in Listing 7.

Listing 7. Input redirection

```
[ian@echidna lpi103-4]$ tr ' ' '\t'<text1
1       apple
2       pear
3       banana
```

Shells, including bash, also have the concept of a *here-document*, which is another form of input redirection. This uses the << along with a word, such as END, for a marker or sentinel to indicate the end of the input. We illustrate this in Listing 8.

Listing 8. Input redirection with a here-document

```
[ian@echidna lpi103-4]$ sort -k2 <<END
> 1 apple
> 2 pear
> 3 banana
> END
1 apple
3 banana
2 pear
```

You may wonder if you couldn't have just typed `sort -k2`, entered your data, and then pressed **Ctrl-d** to signal end of input. The short answer is that you could, but you would not have learned about here-documents. The real answer is that here-documents are more often used in shell scripts. (A script doesn't have any other way of signaling which lines of the script should be treated as input.) Because shell scripts make extensive use of tabbing to provide indenting for readability, there is another twist to here-documents. If you use <<- instead of just <<, then leading tabs are stripped.

In Listing 9 we create a captive tab character using command substitution and then create a very small shell script containing two `cat` commands, which each read from a here-document. Note that we use END as the sentinel for the here-document we are reading from the terminal. If we used the same word for the sentinel within the script, we would end our typing prematurely. So we use EOF instead. After our script is created, we use the `.` (dot) command to *source* it, which means to run it in the current shell context.

Listing 9. Input redirection with a here-document

```
[ian@echidna lpi103-4]$ ht=$(echo -en "\t")
[ian@echidna lpi103-4]$ cat<<END>ex-here.sh
> cat <<-EOF
> apple
> EOF
> ${ht}cat <<-EOF
> ${ht}pear
> ${ht}EOF
> END
[ian@echidna lpi103-4]$ cat ex-here.sh
cat <<-EOF
apple
EOF
        cat <<-EOF
        pear
        EOF
[ian@echidna lpi103-4]$ . ex-here.sh
apple
pear
```

You will learn more about command substitution and scripting in later articles of this series. See our series roadmap for a description of and link to each article in the series.

Creating pipelines

In the article "Learn Linux 101: Text streams and filters," we described text *filtering* as the process of taking an input stream of text and performing some conversion on the text before sending it to an output stream. Such filtering is most often done by constructing a *pipeline* of commands where the output from one command is *piped* or *redirected* to be used as input to the next. Using pipelines in this way is not restricted to text streams, although that is often where they are used.

Piping stdout to stdin

You use the | (pipe) operator between two commands to direct the stdout of the first to the stdin of the second. You construct longer pipelines by adding more commands and more pipe operators. Any of the commands may have options or arguments. Many commands use a hyphen (-) in place of a filename as an argument to indicate when the input should come from stdin rather than a file. Check the man pages for the command to be sure. Constructing long pipelines of commands that each have limited capability is a common Linux and UNIX® way of accomplishing tasks. In the hypothetical pipeline in Listing 10, `command2` and `command3` both have parameters, while `command3` uses the `-` parameter alone to signify input from stdin.

Listing 10. Piping output through several commands

```
command1 | command2 paramater1 | command3 parameter1 - parameter2 | command4
```

One thing to note is that pipelines **only** pipe stdout to stdin. You cannot use 2| to pipe stderr alone, at least, not with the tools we have learned so far. If stderr has been redirected to stdout, then both streams will be piped. In Listing 11 we illustrate an unlikely `ls` command with four wildcard arguments that are not in alphabetical order, then use a pipe sort the combined normal and error output.

Listing 11. Piping two output streams

```
[ian@echidna lpi103-4]$ ls y* x* z* u* q*
ls: cannot access z*: No such file or directory
ls: cannot access u*: No such file or directory
ls: cannot access q*: No such file or directory
xaa  xab  yaa  yab
[ian@echidna lpi103-4]$ ls y* x* z* u* q*  2>&1 |sort
ls: cannot access q*: No such file or directory
ls: cannot access u*: No such file or directory
ls: cannot access z*: No such file or directory
xaa
xab
yaa
yab
```

One advantage of pipes on Linux and UNIX systems is that, unlike some other popular operating systems, there is no intermediate file involved with a pipe. The stdout of the first command is **not** written to a file and then read by the second command. In the article "Learn Linux, 101: File and directory management," you learn how to archive and compress a file in one step using the `tar` command. If you happen to be working on a UNIX system where the `tar` command does not support compression using the `-z` (for gzip) or `-j` (for bzip2) compression, that's no problem. You can just use a pipeline like

```
bunzip2 -c somefile.tar.bz2 | tar -xvf -
```

to do the task.

Starting pipelines with a file instead of stdout

In the above pipelines we started with some command that generated output and then piped that output through each stage of the pipeline. What if we want to start with a file of data that already exists? Many commands take either stdin or a file as input, so those aren't a problem. If you do have a filter that requires input from stdin, you might think of using the `cat` command to copy the file to stdout, which would work. However, you can use input redirection for the first command and then pipe that command's output through the rest of the pipeline for the more usual solution. Just use the < operator to redirect the stdin of your first command to the file you want to process.

Using output as arguments

In the preceding discussion of pipelines, you learned how to take the output of one command and use it as input to another command. Suppose, instead, that you want to use the output of a command or the contents of a file as arguments to a command rather than as input. Pipelines don't work for that. Three common methods are:

1. The `xargs` command
2. The `find` command with the `-exec` option
3. Command substitution

You will learn about the first two of these now. You saw an example of command substitution in Listing 9 when we created a captive tab character. Command substitution is used at the command line, but more frequently in scripting; you will learn more about it and scripting in later articles of this series. See our series roadmap for a description of and link to each article in the series.

Using the `xargs` command

The `xargs` command reads standard input and then builds and executes commands with the input as parameters. If no command is given, then the `echo` command is used. Listing 12 is a basic example using our text1 file, which contains three lines each having two words..

Listing 12. Using `xargs`

```
[ian@echidna lpi103-4]$ cat text1
1 apple
2 pear
3 banana
[ian@echidna lpi103-4]$ xargs<text1
1 apple 2 pear 3 banana
```

So why is there only one line of output from `xargs`? By default, `xargs` breaks the input at blanks, and each resulting token becomes a parameter. However, when `xargs` builds the command, it will pass as many parameters at once as it possibly can. You may override this behavior with the `-n`, or `--max-args`, parameter. In Listing 13 we illustrate the use of both forms and add an explicit invocation of `echo` to our use of `xargs`.

Listing 13. Using `xargs` and `echo`

```
[ian@echidna lpi103-4]$ xargs<text1 echo "args >"
args > 1 apple 2 pear 3 banana
[ian@echidna lpi103-4]$ xargs --max-args 3 <text1 echo "args >"
args > 1 apple 2
args > pear 3 banana
[ian@echidna lpi103-4]$ xargs -n 1 <text1 echo "args >"
args > 1
args > apple
args > 2
args > pear
args > 3
args > banana
```

If the input contains blanks that are protected by single or double quotes, or by backslash escaping, then `xargs` will not break the input at such points. Listing 14 illustrates this point.

Listing 14. Using `xargs` with quoting

```
[ian@echidna lpi103-4]$ echo '"4 plum"' | cat text1 -
1 apple
2 pear
3 banana
"4 plum"
[ian@echidna lpi103-4]$ echo '"4 plum"' | cat text1 - | xargs -n 1
1
apple
2
pear
3
banana
4 plum
```

So far, all the arguments have been added to the end of the command. If you have a requirement to use them as arguments with other arguments following, then you use the `-I` option to specify a replacement string. Wherever the replacement string occurs in the command you ask `xargs` to

execute, it will be replaced by an argument. When you do this, only one argument will be passed to each command. However, the argument will be created from a whole line of input, not just a single token. You can also use the `-L` option of `xargs` to have it treat lines as arguments rather than the default of individual blank-delimited tokens. Using the `-I` option implies `-L 1`. Listing 15 shows examples of the use of both `-I` and `-L`

Listing 15. Using `xargs` with lines of input

```
[ian@echidna lpi103-4]$ xargs -I XYZ echo "START XYZ REPEAT XYZ END" <text1
START 1 apple REPEAT 1 apple END
START 2 pear REPEAT 2 pear END
START 3 banana REPEAT 3 banana END
[ian@echidna lpi103-4]$ xargs -IX echo "<X><X>" <text2
<9      plum><9 plum>
<3      banana><3       banana>
<10     apple><10       apple>
[ian@echidna lpi103-4]$ cat text1 text2 | xargs -L2
1 apple 2 pear
3 banana 9 plum
3 banana 10 apple
```

Although our examples have used simple text files for illustration, you will seldom want to use `xargs` with input like this. Usually you will be dealing with a large list of files generated from a command such as `ls`, `find`, or `grep`. Listing 16 shows one way you might pass a directory listing through `xargs` to a command such as `grep`.

Listing 16. Using `xargs` with lists of files

```
[ian@echidna lpi103-4]$ ls |xargs grep "1"
text1:1 apple
text2:10        apple
xaa:1 apple
yaa:1
```

What happens in the last example if one or more of the file names contains a space? If you just used the command as in Listing 16, then you would get an error. In the real world, your list of files may come from some source such as a custom script or command, rather than `ls`, or you may wish to pass it through some other pipeline stages to further filter it, so we'll ignore the fact that you could just use `grep "1" *` instead of this construct.

For the `ls` command, you could use the `--quoting-style` option to force the problem file names to be quoted or escaped. A better solution, when available, is the `-0` option of `xargs`, so that the null character (\0) is used to delimit input arguments. Although `ls` does not have an option to provide null-terminated file names as output, many commands do.

In Listing 17 we first copy text1 to "text 1" and then show some ways of using a list of file names containing blanks with `xargs`. These are for illustration of the concepts, as `xargs` can be tricky to master. In particular, the final example of translating new line characters to nulls would not work if some file names already contained new line characters. In the next part of this article, we'll look at a more robust solution using the `find` command to generate suitable null-delimited output.

Listing 17. Using `xargs` with blanks in file names

```
[ian@echidna lpi103-4]$ cp text1 "text 1"
[ian@echidna lpi103-4]$ ls *1 |xargs grep "1" # error
text1:1 apple
grep: text: No such file or directory
grep: 1: No such file or directory
[ian@echidna lpi103-4]$ ls --quoting-style escape *1
text1  text\ 1
[ian@echidna lpi103-4]$ ls --quoting-style shell *1
text1  'text 1'
[ian@echidna lpi103-4]$ ls --quoting-style shell *1 |xargs grep "1"
text1:1 apple
text 1:1 apple
[ian@echidna lpi103-4]$ # Illustrate -0 option of xargs
[ian@echidna lpi103-4]$ ls *1 | tr '\n' '\0' |xargs -0 grep "1"
text1:1 apple
text 1:1 apple
```

The `xargs` command will not build arbitrarily long commands. Until Linux kernel 2.26.3, the maximum size of a command was limited. A command such as `rm somepath/*`, for a directory containing a lot of files with long names, might fail with a message indicating that the argument list was too long. In the older Linux systems, or on UNIX systems that may still have the limitation, it is useful to know how to use `xargs` so you can handle such situations

You can use the `--show-limits` option to display the default limits of `xargs`, and the `-s` option to limit the size of output commands to a specific maximum number of characters. See the man pages for other options that we have not discussed here.

Using the `find` command with the `-exec` option or with `xargs`

In the article "Learn Linux, 101: File and directory management," you learn how to use the `find` command to find files by name, modification time, size, or other characteristics. Once you find such a set of files, you will often want to do something with them: remove them, copy them to another location, rename them, or some other operation. Now we will look at the `-exec` option of `find`, which has similar functionality to using `find` and piping the output to `xargs`.

Listing 18. Using `find` and `-exec`

```
[ian@echidna lpi103-4]$ find text[12] -exec cat text3 {} \;
This is a sentence.  This is a sentence.  This is a sentence.
1 apple
2 pear
3 banana
This is a sentence.  This is a sentence.  This is a sentence.
9       plum
3       banana
10      apple
```

Compared to what you already learned about using `xargs`, there are several differences.

1. You **must** include the {} to mark where the filename goes in the command. It is not automatically added at the end.
2. You must terminate the command with a semi-colon and it must be escaped; \;, ';', or ";" will do.

3. The command is executed once for each input file.

Try running `find text[12] |xargs cat text3` to see the differences for yourself.

Now let's return to the case of the blank in the file name. In Listing 19 we try using `find` with `-exec` rather than `ls` with `xargs`.

Listing 19. Using `find` and `-exec` with blanks in file names

```
[ian@echidna lpi103-4]$ find . -name "*1" -exec grep "1" {} \;
1 apple
1 apple
```

So far, so good. But isn't there something missing? Which files contained the lines found by `grep`? The file name is missing because `find` called `grep` once for each file, and `grep` is smart enough to know that if you only give it one file name, then you don't need it to tell you what file it was.

We could use `xargs` instead, but we already saw problems with blanks in file names. We also alluded to the fact that `find` could produce a list of file names with null delimiters, and that is what the `-print0` option does. Modern versions of find may be delimited with a + sign instead of a semi-colon, and this causes find to pass as many names as possible in one invocation of a command, similar to the way `xargs` works. Needless to say, you may only use {} once in this case, and it must be the last parameter of the command. Listing 20 shows both methods.

Listing 20. Using `find` and `xargs` with blanks in file names

```
[ian@echidna lpi103-4]$ find . -name "*1" -print0 |xargs -0 grep "1"
./text 1:1 apple
./text1:1 apple
[ian@echidna lpi103-4]$ find . -name "*1" -exec grep "1" {} +
./text 1:1 apple
./text1:1 apple
```

Generally, either method will work, and the choice is often a matter of personal style. Remember that piping things with unprotected blanks or white space may cause problems, so use the `-print0` option with `find` if you are piping the output to `xargs`, and use the `-0` option to tell `xargs` to expect null-delimited input. Other commands, including `tar`, also support null-delimited input using the `-0` option, so you should always use it for commands that support it unless you are certain that your input list will not be a problem.

One final comment on operating on a list of files. It's always a good idea to thoroughly test your list and also to test your command very carefully before committing to a bulk operation such as removing or renaming files. Having a good backup can be invaluable too.

Splitting output

This section wraps up with a brief discussion of one more command. Sometimes you may want to see output on your screen while saving a copy for later. While you **could** do this by redirecting the command output to a file in one window and then using `tail -fn1` to follow the output in another screen, using the `tee` command is easier.

You use `tee` with a pipeline. The arguments are a file (or multiple files) for standard output. The `-a` option appends rather than overwriting files. As we saw earlier in our discussion of pipelines, you need to redirect stderr to stdout before piping to `tee` if you want to save both. Listing 21 shows `tee` being used to save output in two files, f1 and f2.

Listing 21. Splitting stdout with tee

```
[ian@echidna lpi103-4]$ ls text[1-3]|tee f1 f2
text1
text2
text3
[ian@echidna lpi103-4]$ cat f1
text1
text2
text3
[ian@echidna lpi103-4]$ cat f2
text1
text2
text3
```

Resources

Learn

- Develop and deploy your next app on the IBM Bluemix cloud platform.
- Use the developerWorks roadmap for LPIC-1 to find the developerWorks articles to help you study for LPIC-1 certification based on the April 2009 objectives.
- At the LPIC Program site, find detailed objectives, task lists, and sample questions for the three levels of the Linux Professional Institute's Linux system administration certification. In particular, see their April 2009 objectives for LPI exam 101 and LPI exam 102. Always refer to the LPIC Program site for the latest objectives.
- Review the entire LPI exam prep series on developerWorks to learn Linux fundamentals and prepare for system administrator certification based on earlier LPI exam objectives prior to April 2009.
- In "Basic tasks for new Linux developers" (developerWorks, March 2005), learn how to open a terminal window or shell prompt and much more.
- The Linux Documentation Project has a variety of useful documents, especially its HOWTOs.
- In the developerWorks Linux zone, find more resources for Linux developers, and scan our most popular articles and tutorials.
- See all Linux tips and Linux tutorials on developerWorks.
- Stay current with developerWorks technical events and Webcasts.

Get products and technologies

- With IBM trial software, available for download directly from developerWorks, build your next development project on Linux.

Discuss

- Participate in the discussion forum for this content.
- Get involved in the My developerWorks community; with your personal profile and custom home page, you can tailor developerWorks to your interests and interact with other developerWorks users.

About the author

Ian Shields

Ian Shields works on a multitude of Linux projects for the developerWorks Linux zone. He is a Senior Programmer at IBM at the Research Triangle Park, NC. He joined IBM in Canberra, Australia, as a Systems Engineer in 1973, and has since worked on communications systems and pervasive computing in Montreal, Canada, and RTP, NC. He has several patents and has published several papers. His undergraduate degree is in pure mathematics and philosophy from the Australian National University. He has an M.S. and Ph.D. in computer science from North Carolina State University. Learn more about Ian in Ian's profile on developerWorks Community.