Connections

by Brian D. Ripley

Connections are a new concept in version 1.2.0 and still in an embryonic state, with some more facilities being made available in version 1.2.1. They are a far-reaching replacement for file-oriented input/output. They are modelled on the facilities introduced in S version 4 and described in Chambers (1998) but are likely to diverge away from that model as they become more central to R.

Three recent enquiries to `r-help` illustrate why one might want something more general than files. David Firth wanted to grab the output from an R function and send it via a socket to a machine in London (in connection with election forecasting as and when there comes a UK general election). He was happy to grab the output to a character vector, check it and then send it using `make.socket` and `write.socket`. Until version 1.2.0 the carefully formatted output of the standard statistical analysis functions could only be sent to the console or a file *via* `sink`. Using *output text connections*, `sink` can now 'trickle' output to a character vector.

Peter Kleiweg had a dataset which was comma-separated, but also had commas at the ends of lines. That might be good for human readers, but is not the format required by `scan`. My proffered solution was to use a *pipe connection* to pre-process the data file whilst it was being read into R, for example

```
zz <- pipe("sed -e s/,$// data")
res <- scan(zz, sep=",")
close(zz)
```

Erich Neuwirth wanted to remotely control R via a socket. That is, a read-mode socket could be created in an R session that would receive R expression as if typed at the command line and process each expression as soon as it was completed. We cannot do exactly that yet (although it was already planned), but one could set up a *pipe connection* to an external program (say `fromsocket`) to read from the socket and use a loop containing something like

```
zz <- pipe("fromsocket socket.id", "r")
repeat {
  ## read 1 expression
  eval(parse(file=zz, n=1))
  ## deal with other things here
  if(some condition) break;
}
close(zz)
```

although of course one needs to take care of error conditions, perhaps by using `try`.

The third example shows one of the main advantages of connections: they can be left open and read from (or written to) repeatedly. Previously the only way to repeatedly read from a file was to keep track of the number of rows read and use the `skip` argument to `scan` and allies. For writing, `cat` had an `append` argument to write to the end of a file. This was all tedious, and various functions (such as `sink` and `dump`) gained `append` arguments as users found a need for them. It was also expensive to keep on closing and re-opening files, especially if they were mounted from remote servers. There were other dangers, too. Suppose we were accessing a file and some process unlinked the file.[3] Then once the file is closed in R it will vanish. Unlinking the file could be accidental, but it is also a very sensible way to provide some protection, both against other processes interfering with it and against leaving it around after we have finished with it.

Types of connections

At present we have connections of types

file A text or binary file, for reading, writing or appending.

terminal There are three standard connections which are always available and always open, called `stdin`, `stdout` and `stderr`. These essentially refer to the corresponding C file names, so `stdin` is input from the console (unless input has been redirected when R was launched) whereas `stdout` and `stderr` are normally both output to the console.

pipe Used for either reading or writing. There is a special subclass for Windows GUI applications (not `rterm`) as the standard C pipes do not work there.

text An R character vector used as a text source.

output text A means to write text output to an R character vector, with each line forming a new element of the vector. Lines become visible as part of the vector in R immediately they are complete.

We envisage adding types. My scene-setting examples illustrated the potential usefulness of sockets as connections. There has been a proliferation of ways to access URIs,[4] such as `read.table.url`. Many (including this one) 'cheat' by downloading a file to a temporary location and then reading from that file. For `read.table`[5] that is not a bad solution,

[3] yes, that works under Unix at least.
[4] more commonly but less accurately known as URLs
[5] which currently needs to read the file twice

but it is desirable for `read.table` to be able to at least *appear* to read from connections that are URIs, and it looks desirable for R to have some basic abilities to read from `http` and `ftp` servers.

Good citizenship

A connection can exist and not be open, the life cycle of most types of connections being

$$\text{create} \longrightarrow \text{open} \longleftrightarrow \text{close} \longrightarrow \text{destroy}$$

A connection is created by its constructor, for example `file` or `pipe`, which can optionally open it. The 'good citizen' rule is that a function which finds a connection open should leave it open, but one that needs to open it should close it again. Thus the life cycle can include multiple passes across the double-sided arrow.

One has to be careful with the terminology here. With one exception, a function internally never destroys a connection, but the function `close` both closes (if necessary) and destroys one. The exception is `sink`, which does close and destroy the current sink connection, unless it is a terminal connection. There is no way for a user to explicitly close a connection without destroying it.

Text connections are special: they are open from creation until destruction. 'Closing' an output text connection flushes out any final partial line of output. (You can call `isIncomplete` to see if there is one.)

These semantics may change. R version 1.2.0 also introduced *references*, which allow finalization actions for referents once there is no R object referring to them, and this may be used to avoid the need for the explicit destroying of connections.

Leaving a connection closed but not destroyed is a minor waste of resources. (There is a finite set of connections and each takes a little memory.) One can find out about all the connections by `showConnections(all = TRUE)`, and there is a function `closeAllConnections` to close all close-able connections (that is, not the terminal connections).

What can I do with connections?

In short, almost everything you used to do with files. The exceptions are the graphics devices which write to files, and it is not yet clear if it is beneficial to be able to send graphics output to connections. The most obvious uses, to pipe the file to a printer, are already possible via other mechanisms.

There are new things made possible by the notion of leaving a connection open. For text-oriented applications, `readLines` and `writeLines` can read and write limited or unlimited amounts of lines from and to a character vector. Many connections are seekable (check this by `isSeekable`) and so can be restored to an arbitrary position.[6]

One can also for the first time work with binary files in base R.[7] Functions `readBin` and `writeBin` can read and write R vector objects (excluding lists) to a connection in binary mode. There are options to assist files to be transferred from or to other programs and platforms, for example to select the endian-ness and the size of the storage type.

Text *vs* binary

There is a distinction between text mode and binary mode connections. The intention is that text-based functions like `scan` and `cat` should use text mode connections, and binary mode is used with `readBin` and `writeBin`.

This distinction is not yet consistently enforced, and the main underlying difference is whether files are opened in text or binary mode (where that matters). It now looks as if opening all files in binary mode and managing the translation of line endings internally in R will lead to fewer surprises. Already reading from a connection in text mode translates lines endings from Unix (LF), DOS/Windows (CRLF) and Macintosh (CR) formats (and from all connections, not just files).

Looking forwards

I see connections as playing a more central rôle in future releases of R. They are one way to promote distributed computing, especially where a stream-oriented view rather than an object-based view is most natural.

The details are likely to change as we gain experience with the ways to use connections. If you rely on side-effects, it is well to be aware that they may change, and the best advice is to manage the connections yourself, explicitly opening and destroying them.

Reference

Chambers, J. M. (1998) *Programming with Data. A Guide to the S Language.* Springer-Verlag.

Brian D. Ripley
University of Oxford, UK
`ripley@stats.ox.ac.uk`

[6]at least if the underlying OS facilities work correctly, which they appear not to for Windows text files.
[7]Package **Rstreams** has provided a way to do so, and is still slightly more flexible.