SciViews socket server

Philippe Grosjean (phgrosjean@sciviews.org)

2024-02-29

The SciViews {svSocket} server implements an R server that is mainly designed to interact with the R prompt from a separate process. The {svSocket} clients and the R console share the same global environment (`.GlobalEnv`). So, everybody interacts with the same variables. Use cases are to build a separate R console, to monitor or query an R session, or even, to build a complete GUI or IDE on top of R. Examples of such GUIs are Tinn-R and Komodo IDE with the SciViews-K extension.

Quick install and use

The SciViews {svSocket} package provides a stateful, multi-client and preemptive socket server. Socket transactions are operational even when R is busy in its main event loop (calculation done at the prompt). This R socket server uses the excellent asynchronous socket ports management by Tcl, and thus, it needs a working version of Tcl/Tk (>= 8.4), the {tcltk} R package, and R with `isTRUE(capabilities("tcltk"))`.

Install the {svSocket} package as usual:

```
install.packages("svSocket")
```

(For the development version, install first `devtools` or `remotes`, and then use something like `remotes::github_install("SciViews/svSocket")`)

```
# Start a separate R process with a script that launch a socket server on 8889
# and wait for the variable `done` in `.GlobalEnv` to finish
rscript <- Sys.which("Rscript")
system2(rscript, "--vanilla -e 'svSocket::start_socket_server(8889); while (!exists(\"done\")) Sy
```

Sometimes (on MacOS Catalina or more, for instance), you are prompted to allow for R to use a socket. Of course, you have to allow to get an operational server.

What this code does is:

- Get the path to `Rscript` using `Sys.which` -Start it with a command that launches the socket server on port 8889[1] (`svSocket::start_socket_server(8889)`)
- Wait that the variable `done` appears in `.GlobalEnv` as a simple mean to keep the server alive.
- Since `system2()` is invoked with `wait = FALSE`, the command exits as soon as the server is created and you can interact at the R prompt.

As you can see, creating a socket server with {svSocket} is easy. Now, let's connect to it. It is also very simple:

```
con <- socketConnection(host = "localhost", port = 8889,
  blocking = FALSE, timeout = 30)
```

Here, we use the `socketConnection()` that comes with base R in non-blocking mode and a time out at 30 sec. So, {svSocket} is even not required to connect an R client to our server. The connection is not blocking. We have the control of the prompt immediately.

Now, we can execute code in our R server process with `svSocket::eval_socket_server()`:

```
library(svSocket)
eval_socket_server(con, '1 + 1')
```

The instruction is indeed executed in the server, and the result is returned to the client transparently. You now master two separate R process: the original one where you interact at the R prompt (›), and the server process that you can interact with through `eval_socket_server()`. To understand this, we will create the variable `x` on both processes, but with different values:

```
# Local x
x <- "local"
# x on the server
eval_socket_server(con, 'x <- "server"')
```

Now, let's look what is available on the server side:

```
eval_socket_server(con, 'ls()')
eval_socket_server(con, 'x')
```

Obviously, there is only one variable, `x`, with value `"server"`. All right ... and in our original process?

```
ls()
x
```

We have `x`, but also `con` and `rsscript`. The value of `x` locally is `"local"`.

As you can see, commands and results are rather similar. And since the processes are different, so are `x` in both processes.

If you want to transfer data to the server, you can still use `eval_socket_server()`, with the name you want for the variable on the server instead of a string with R code, and as third argument, the name of the local variable whose the content will be copied to the server. `eval_socket_server()` manages to send the data to the server (by serializing the data on your side and deserializing it on the server). Here, we will transfer the `iris` data frame to the server under the `iris2` name:

```
data(iris)
eval_socket_server(con, iris2, iris)
eval_socket_server(con, "ls()")          # iris2 is there
eval_socket_server(con, "head(iris2)") # ... and its content is OK
```

For more examples on using `eval_socket_server`, see its man page with `?eval_socket_server`.

Lower-level interaction

The {svSocket} server also allows, of course, for a lower-level interaction with the server, with a lot more options. Here, you send something to the server using `cat, file = con)` (make sure to end your command by a carriage return `\n`, or the server will wait indefinitely for the next part of the instruction!), and you read results using `readLines(con)`. Of course, you must wait that R processes the command before reading results back:

```
# Send a command to the R server (low-level version)
cat("{Sys.sleep(2); "Done!"}\n", file = con)
# Wait for, and get response from the server
res <- NULL
while (!length(res)) {
  Sys.sleep(0.01)
  res <- readLines(con)
}
res
```

Here you got the results send back as strings. If you want to display them on the client's console as if it was output there (like `eval_socket_server()` does), you should use `cat(, sep = "\n")`:

```
cat(res, "\n")
```

For convenience, we will wrap all this a function `run_socket_server()`:

```
run_socket_server <- function(con, code) {
  cat(code, "\n", file = con)
  res <- NULL
  while (!length(res)) {
    Sys.sleep(0.01)
    res <- readLines(con)
  }
  # Use this instruction to output results as if code was run at the prompt
  #cat(res, "\n")
  invisible(res)
}
```

The transaction is now much easier:

```
(run_socket_server(con, '{Sys.sleep(2); "Done!"}'))
```

Configuration and special instructions

Now at the low-level (not within `eval_socket_server()` but within our `run_socket_server()` function), one can insert special code `<<<X>>>` with `X` being a marker that the server will use on his side. The last instruction we send instructed the server to wait for 2 sec and then, to return `"Done!"`. We have send the instruction to the server and *wait* for it to finish processing and then, we got the results back. Thus, you original R process is locked down the time the server processes the code. Note that we had to lock it down on our side using the `while(!length(res))` construct. It means that communication between the server and the client is asynchronous, but process of the command must be made synchronous. If you want to return immediately in the calling R process *before* the instruction is processed in the server, you could just consider to drop the `while(...)` section. **This is not a good idea!** Indeed, R will send results back and you will read them on the next `readLines(con)` you issue, and mix it with, perhaps, the result of the next instruction. So, here, we really must tell the R server to send nothing back to us. This is done by inserting the special instruction `<<<h>>>` on one line (surrounded by `\n`). This way, we still have to wait for the instruction to be processed on the server, but nothing is returned back to the client. Also, sending `<<<h>>>` will result into an immediate finalization of the transaction by the server *before* the instruction is processed.

```
(run_socket_server(con, '\n<<<h>>>{Sys.sleep(2); "Done!"}'))
```

Here, we got the result immediately, but it is *not* the results of the code execution. Our R server simply indicates that he got our code, he parsed it and is about to process it on his side by returning an empty string `""`.

There are several special instructions you can use. See `?par_socket_server` for further details. The server can be configured to behave in a given way, and that configuration is persistent from one connection to the other for the same client. The function `par_socket_server()` allows to change or query the configuration. Its first argument is the name of the client on the server-side... but from the client, you don't necessarily know which name the server gave to you (one can connect several different clients at the same time, and default name is `sock` followed by Tcl name of the client socket connection). Using `<<<s>>>` as a placeholder for this name circumvents the problem. `par_socket_server()` returns an environment that contains configuration variables. Here is the list of configuration item for us:

```
cat(run_socket_server(con, 'ls(envir = svSocket::par_socket_server(<<<s>>>))'), sep = "\n")
```

The `bare` item indicates if the server sends bare results, or also returns a prompt, acting more like a terminal. By default, it returns the bare results. Here is the current value:

```
cat(run_socket_server(con, 'svSocket::par_socket_server(<<<s>>>)$bare'), sep = "\n")
```

And here is how you can change it:

```
(run_socket_server(con, '\n<<<h>>>svSocket::par_socket_server(<<<s>>>, bare = FALSE)'))
(run_socket_server(con, '1 + 1'))
```

When `bare = FALSE` the server issues the formatted command with a prompt (`:>` by default) and the result. For more information about {svSocket} server configuration, see `?par_socket_server`. There are also functions to manipulate the pool of clients and their configurations from the server-side, and the server can also send unattended data to client, see `?send_socket_clients`. finally, the workhorse function on the server-side is `process_socket_server()`. See `?process_socket_server` to learn more about it. You can provide your own process function to the server if you need to.

Disconnection

Don't forget to close the connection, once you have done using `close(con)`, and you can also close the server from the server-side by using `stop_socket_server()`. But never use it from the client-side because you will obviously break in the middle of the transaction. If you want to close the server from the client, you have to install a mechanisms that will nicely shut down the server *after* the transaction is processed. Here, we have installed such a mechanism by detecting the presence of a variable named `done` on the server-side. So:

```
# Usually, the client does not stop the server, but it is possible here
eval_socket_server(con, 'done <- NULL') # The server will stop after this transaction
close(con)
```

You can also access the {svSocket} server from another language. There is a very basic example written in Tcl in the `/etc` subdirectory of the {svSocket} package. See the `ReadMe.txt` file there. The Tcl script `SimpleClient.tcl` implements a Tcl client in a few dozens of code lines. For other examples, you can inspect the code of SciViews-K, and the code of Tinn-R for a Pascal version. Writing clients in C, Java, Python, etc. should not be difficult if you inspire you from these examples. Finally, there is another implementation of a similar R server through HTTP in the {svHttp} package.

---

1. Of course, this port must be free. If not, just use another value.↩