A Future for R: A Comprehensive Overview

Introduction

The purpose of the future package is to provide a very simple and uniform way of evaluating R expressions asynchronously using various resources available to the user.

In programming, a *future* is an abstraction for a *value* that may be available at some point in the future. The state of a future can either be *unresolved* or *resolved*. As soon as it is resolved, the value is available instantaneously. If the value is queried while the future is still unresolved, the current process is *blocked* until the future is resolved. It is possible to check whether a future is resolved or not without blocking. Exactly how and when futures are resolved depends on what strategy is used to evaluate them. For instance, a future can be resolved using a sequential strategy, which means it is resolved in the current R session. Other strategies may be to resolve futures asynchronously, for instance, by evaluating expressions in parallel on the current machine or concurrently on a compute cluster.

Here is an example illustrating how the basics of futures work. First, consider the following code snippet that uses plain R code:

```
> v <- {
+   cat("Resolving...\n")
+   3.14
+ }
Resolving...
> v
[1] 3.14
```

It works by assigning the value of an expression to variable `v` and we then print the value of `v`. Moreover, when the expression for `v` is evaluated we also print a message.

Here is the same code snippet modified to use futures instead:

```
> library("future")
> v %<-% {
+   cat("Resolving...\n")
+   3.14
+ }
Resolving...
> v
[1] 3.14
```

The difference is in how `v` is constructed; with plain R we use `<-` whereas with futures we use `%<-%`.

So why are futures useful? Because we can choose to evaluate the future expression in a separate R process asynchronously by simply switching settings as:

```
> library("future")
> plan(multiprocess)
> v %<-% {
+   cat("Resolving...\n")
+   3.14
+ }
> v
[1] 3.14
```

With asynchronous futures, the current/main R process does *not* block, which means it is available for further processing while the futures are being resolved in separate processes running in the background. In other words, futures provide a simple but yet powerful construct for parallel and / or distributed processing in R.

Now, if you cannot be bothered to read all the nitty-gritty details about futures, but just want to try them out, then skip to the end to play with the Mandelbrot demo using both parallel and non-parallel evaluation.

Implicit or Explicit Futures

Futures can be created either *implicitly* or *explicitly*. In the introductory example above we used *implicit futures* created via the `v %<-% { expr }` construct. An alternative is *explicit futures* using the `f <- future({ expr })` and `v <- value(f)` constructs. With these, our example could alternatively be written as:

```
> library("future")
> f <- future({
+   cat("Resolving...\n")
+   3.14
+ })
Resolving...
> v <- value(f)
> v
[1] 3.14
```

Either style of future construct works equally(*) well. The implicit style is most similar to how regular R code is written. In principle, all you have to do is to replace `<-` with a `%<-%` to turn the assignment into a future assignment. On the other hand, this simplicity can also be deceiving, particularly when asynchronous futures are being used. In contrast, the explicit style makes it much clearer that futures are being used,

which lowers the risk for mistakes and better communicates the design to others reading your code.

(*) There are cases where `%<-%` cannot be used without some (small) modifications. We will return to this in Section 'Constraints when using Implicit Futures' near the end of this document.

To summarize, for explicit futures, we use:

- `f <- future({ expr })` - creates a future
- `v <- value(f)` - gets the value of the future (blocks if not yet resolved)

For implicit futures, we use:

- `v %<-% { expr }` - creates a future and a promise to its value

To keep it simple, we will use the implicit style in the rest of this document, but everything discussed will also apply to explicit futures.

Controlling How Futures are Resolved

The future package implements the following types of futures:

Name	OSes	Description
synchronous:		*non-parallel:*
`sequential`	all	sequentially and in the current R process
`transparent`	all	as sequential w/ early signaling and w/out local (for debugging)
asynchronous:		*parallel*:
`multiprocess`	all	multicore, if supported, otherwise multisession
`multisession`	all	background R sessions (on current machine)
`multicore`	not Windows	forked R processes (on current machine)
`cluster`	all	external R sessions on current, local, and/or remote machines
`remote`	all	Simple access to remote R sessions

The future package is designed such that support for additional strategies can be implemented as well. For instance, the future.batchtools package provides futures for all types of *cluster functions* (“backends”) that the batchtools package supports. Specifically, futures for evaluating R expressions via job schedulers such as Slurm, TORQUE/PBS, Oracle/Sun Grid Engine (SGE) and Load Sharing Facility (LSF) are also available. (*Comment*: The future.BatchJobs package provides analogue backends based on the BatchJobs package; however the BatchJobs developers have deprecated it in favor of batchtools.)

By default, future expressions are evaluated eagerly (= instantaneously) and synchronously (in the current R session). This evaluation strategy is referred to as “sequential”. In this section, we will go through each of these strategies and discuss

what they have in common and how they differ.

Consistent Behavior Across Futures

Before going through each of the different future strategies, it is probably helpful to clarify the objectives the Future API (as defined by the future package). When programming with futures, it should not really matter what future strategy is used for executing code. This is because we cannot really know what computational resources the user has access to so the choice of evaluation strategy should be in the hands of the user and not the developer. In other words, the code should not make any assumptions on the type of futures used, e.g. synchronous or asynchronous.

One of the designs of the Future API was to encapsulate any differences such that all types of futures will appear to work the same. This despite expressions may be evaluated locally in the current R session or across the world in remote R sessions. Another obvious advantage of having a consistent API and behavior among different types of futures is that it helps while prototyping. Typically one would use sequential evaluation while building up a script and, later, when the script is fully developed, one may turn on asynchronous processing.

Because of this, the defaults of the different strategies are such that the results and side effects of evaluating a future expression are as similar as possible. More specifically, the following is true for all futures:

- All *evaluation is done in a local environment* (i.e. `local({ expr })`) so that assignments do not affect the calling environment. This is natural when evaluating in an external R process, but is also enforced when evaluating in the current R session.

- When a future is constructed, *global variables are identified*. For asynchronous evaluation, globals are exported to the R process/session that will be evaluating the future expression. For sequential futures with lazy evaluation (`lazy = TRUE`), globals are “frozen” (cloned to a local environment of the future). Also, in order to protect against exporting too large objects by mistake, there is a built-in assertion that the total size of all globals is less than a given threshold (controllable via an option, cf. `help("future.options")`). If the threshold is exceeded, an informative error is thrown.

- Future *expressions are only evaluated once*. As soon as the value (or an error) has been collected it will be available for all succeeding requests.

Here is an example illustrating that all assignments are done to a local environment:

```
> plan(sequential)
> a <- 1
> x %<-% {
+     a <- 2
+     2 * a
+ }
> x
[1] 4
> a
[1] 1
```

Now we are ready to explore the different future strategies.

Synchronous Futures

Synchronous futures are resolved one after another and most commonly by the R process that creates them. When a synchronous future is being resolved it blocks the main process until resolved. There are two types of synchronous futures in the future package, *sequential* and *transparent*.

Sequential Futures

Sequential futures are the default unless otherwise specified. They were designed to behave as similar as possible to regular R evaluation while still fulfilling the Future API and its behaviors. Here is an example illustrating their properties:

```
> plan(sequential)
> pid <- Sys.getpid()
> pid
[1] 28518
> a %<-% {
+     pid <- Sys.getpid()
+     cat("Resolving 'a' ...\n")
+     3.14
+ }
Resolving 'a' ...
> b %<-% {
+     rm(pid)
+     cat("Resolving 'b' ...\n")
+     Sys.getpid()
+ }
Resolving 'b' ...
> c %<-% {
+     cat("Resolving 'c' ...\n")
+     2 * a
+ }
Resolving 'c' ...
> b
[1] 28518
> c
[1] 6.28
> a
[1] 3.14
> pid
[1] 28518
```

Since eager sequential evaluation is taking place, each of the three futures is resolved instantaneously in the moment it is created. Note also how `pid` in the calling environment, which was assigned the process ID of the current process, is neither overwritten nor removed. This is because futures are evaluated in a local environment. Since synchronous (uni-)processing is used, future `b` is resolved by the main R process (still in a local environment), which is why the value of `b` and `pid` are the same.

Transparent Futures

For troubleshooting, *transparent* futures can be used by specifying `plan(transparent)`. A transparent future is technically a sequential future with instant signaling of conditions (including errors and warnings) and where evaluation, and therefore also assignments, take place in the calling environment. Transparent futures are particularly useful for troubleshooting errors that are otherwise hard to narrow down.

Asynchronous Futures

Next, we will turn to asynchronous futures, which are futures that are resolved in the background. By design, these futures are non-blocking, that is, after being created the calling process is available for other tasks including creating additional futures. It is only when the calling process tries to access the value of a future that is not yet resolved, or trying to create another asynchronous future when all available R processes are busy serving other futures, that it blocks.

Multisession Futures

We start with multisession futures because they are supported by all operating systems. A multisession future is evaluated in a background R session running on the same machine as the calling R process. Here is our example with multisession evaluation:

```
> plan(multisession)
> pid <- Sys.getpid()
> pid
[1] 28518
> a %<-% {
+     pid <- Sys.getpid()
+     cat("Resolving 'a' ...\n")
+     3.14
+ }
> b %<-% {
+     rm(pid)
+     cat("Resolving 'b' ...\n")
+     Sys.getpid()
+ }
> c %<-% {
+     cat("Resolving 'c' ...\n")
+     2 * a
+ }
> b
[1] 28539
> c
[1] 6.28
> a
[1] 3.14
> pid
[1] 28518
```

The first thing we observe is that the values of `a`, `c` and `pid` are the same as previously. However, we notice that `b` is different from before. This is because future `b` is evaluated in a different R process and therefore it returns a different process ID.

Another difference is that the messages, generated by `cat()`, are no longer displayed. This is because they are outputted to the background sessions and not the calling session.

When multisession evaluation is used, the package launches a set of R sessions in the background that will serve multisession futures by evaluating their expressions as they are created. If all background sessions are busy serving other futures, the creation of the next multisession future is *blocked* until a background session becomes available again. The total number of background processes launched is decided by the value of `availableCores()`, e.g.

```
> availableCores()
mc.cores
       2
```

This particular result tells us that the `mc.cores` option was set such that we are allowed to use in total 2 processes including the main process. In other words, with these settings, there will be 2 background processes serving the multisession futures. The `availableCores()` is also agile to different options and system environment variables. For instance, if compute cluster schedulers are used (e.g. TORQUE/PBS and Slurm), they set specific environment variable specifying the number of cores that was allotted to any given job; `availableCores()` acknowledges these as well. If nothing else is specified, all available cores on the machine will be utilized, cf. `parallel::detectCores()`. For more details, please see `help("availableCores", package = "future")`.

Multicore Futures

On operating systems where R supports *forking* of processes, which is basically all operating system except Windows, an alternative to spawning R sessions in the background is to fork the existing R process. Forking an R process is considered faster than working with a separate R session running in the background. One reason is that the overhead of exporting large globals to the background session can be greater than when forking is used. To use multicore futures, we specify:

```
plan(multicore)
```

The only real different between using multicore and multisession futures is that any output written (to standard output or standard error) by a multicore process is instantaneously outputted in calling process. Other than this, the behavior of using multicore evaluation is very similar to that of using multisession evaluation.

Just like for multisession futures, the maximum number of parallel processes running will be decided by `availableCores()`, since in both cases the evaluation is done on the local machine.

Multiprocess Futures

Sometimes we do not know whether multicore futures are supported or not, but it might still be that we would like to write platform-independent scripts or instructions that work everywhere. In such cases we can specify that we want to use “multiprocess” futures as in:

```
plan(multiprocess)
```

A multiprocess future is not a formal class of futures by itself, but rather a convenient alias for either of the two. When this is specified, multisession evaluation will be used unless multicore evaluation is supported.

Cluster Futures

Cluster futures evaluate expressions on an ad-hoc cluster (as implemented by the parallel package). For instance, assume you have access to three nodes `n1`, `n2` and `n3`, you can then use these for asynchronous evaluation as:

```
> plan(cluster, workers = c("n1", "n2", "n3"))
> pid <- Sys.getpid()
> pid
[1] 28518
> a %<-% {
+     pid <- Sys.getpid()
+     cat("Resolving 'a' ...\n")
+     3.14
+ }
> b %<-% {
+     rm(pid)
+     cat("Resolving 'b' ...\n")
+     Sys.getpid()
+ }
> c %<-% {
+     cat("Resolving 'c' ...\n")
+     2 * a
+ }
> b
[1] 28561
> c
[1] 6.28
> a
[1] 3.14
> pid
[1] 28518
```

Just as for most other asynchronous evaluation strategies, the output from `cat()` is not displayed on the current/calling machine.

Any types of clusters that `parallel::makeCluster()` creates can be used for cluster futures. For instance, the above cluster can be explicitly set up as:

```
cl <- parallel::makeCluster(c("n1", "n2", "n3"))
plan(cluster, workers = cl)
```

Also, it is considered good style to shut down cluster `cl` when it is no longer needed, that is, calling `parallel::stopCluster(cl)`. However, it will shut itself down if the main process is terminated. For more information on how to set up and manage such clusters, see `help("makeCluster", package = "parallel")`. Clusters created implicitly using `plan(cluster, workers = hosts)` where `hosts` is a character vector will also be shut down when the main R session terminates, or when the future strategy is changed, e.g. by calling `plan(sequential)`.

Note that with automatic authentication setup (e.g. SSH key pairs), there is nothing preventing us from using the same approach for using a cluster of remote machines.

Nested Futures and Evaluation Topologies

This far we have discussed what can be referred to as “flat topology” of futures, that is, all futures are created in and assigned to the same environment. However, there is nothing stopping us from using a “nested topology” of futures, where one set of futures may, in turn, create another set of futures internally and so on.

For instance, here is an example of two “top” futures (`a` and `b`) that uses multiprocess evaluation and where the second future (`b`) in turn uses two internal futures:

```
> plan(multiprocess)
> pid <- Sys.getpid()
> a %<-% {
+     cat("Resolving 'a' ...\n")
+     Sys.getpid()
+ }
> b %<-% {
+     cat("Resolving 'b' ...\n")
+     b1 %<-% {
+         cat("Resolving 'b1' ...\n")
+         Sys.getpid()
+     }
+     b2 %<-% {
+         cat("Resolving 'b2' ...\n")
+         Sys.getpid()
+     }
+     c(b.pid = Sys.getpid(), b1.pid = b1, b2.pid = b2)
+ }
> pid
[1] 28518
> a
[1] 28584
> b
 b.pid b1.pid b2.pid
 28585  28585  28585
```

By inspection the process IDs, we see that there are in total three different processes involved for resolving the futures. There is the main R process (pid 28518), and there are the two processes used by `a` (pid 28584) and `b` (pid 28585). However, the two futures (`b1` and `b2`) that is nested by `b` are evaluated by the same R process as `b`. This is because nested futures use sequential evaluation unless otherwise specified. There are a few reasons for this, but the main reason is that it protects us from spawning off a large number of background processes by mistake, e.g. via recursive calls.

To specify a different type of *evaluation topology*, other than the first level of futures being resolved by multiprocess evaluation and the second level by sequential evaluation, we can provide a list of evaluation strategies to `plan()`. First, the same evaluation strategies as above can be explicitly specified as:

```
plan(list(multiprocess, sequential))
```

We would actually get the same behavior if we try with multiple levels of multiprocess evaluations;

```
> plan(list(multiprocess, multiprocess))
[...]
> pid
[1] 28518
> a
[1] 28586
> b
 b.pid b1.pid b2.pid
 28587  28587  28587
```

The reason for this is, also here, to protect us from launching more processes than what the machine can support. Internally, this is done by setting `mc.cores = 1` such that functions like `parallel::mclapply()` will fall back to run sequentially. This is the case for both multisession and multicore evaluation.

Continuing, if we start off by sequential evaluation and then use multiprocess evaluation for any nested futures, we get:

```
> plan(list(sequential, multiprocess))
[...]
Resolving 'a' ...
Resolving 'b' ...
> pid
[1] 28518
> a
[1] 28518
> b
 b.pid b1.pid b2.pid
 28518  28588  28589
```

which clearly show that `a` and `b` are resolved in the calling process (pid 28518) whereas the two nested futures (`b1` and `b2`) are resolved in two separate R processes (pids 28588 and 28589).

Having said this, it is indeed possible to use nested multiprocess evaluation strategies, if we explicitly specify (read *force*) the number of cores available at each level. In order to do this we need to “tweak” the default settings, which can be done as follows:

```
> plan(list(tweak(multiprocess, workers = 2L), tweak(multiprocess,
+     workers = 2L)))
[...]
> pid
[1] 28518
> a
[1] 28590
> b
 b.pid b1.pid b2.pid
 28591  28592  28594
```

First, we see that both `a` and `b` are resolved in different processes (pids 28590 and 28591) than the calling process (pid 28518). Second, the two nested futures (`b1` and `b2`) are resolved in yet two other R processes (pids 28592 and 28594).

For more details on working with nested futures and different evaluation strategies at each level, see Vignette 'Futures in R: Future Topologies'.

Checking A Future without Blocking

It is possible to check whether a future has been resolved or not without blocking. This can be done using the `resolved(f)` function, which takes an explicit future `f` as input. If we work with implicit futures (as in all the examples above), we can use the `f <- futureOf(a)` function to retrieve the explicit future from an implicit one. For example,

```
> plan(multiprocess)
> a %<-% {
+     cat("Resolving 'a' ...")
+     Sys.sleep(2)
+     cat("done\n")
+     Sys.getpid()
+ }
> cat("Waiting for 'a' to be resolved ...\n")
Waiting for 'a' to be resolved ...
> f <- futureOf(a)
> count <- 1
> while (!resolved(f)) {
+     cat(count, "\n")
+     Sys.sleep(0.2)
+     count <- count + 1
+ }
1
2
3
4
5
> cat("Waiting for 'a' to be resolved ... DONE\n")
Waiting for 'a' to be resolved ... DONE
> a
[1] 28595
```

Failed Futures

Sometimes the future is not what you expected. If an error occurs while evaluating a future, the error is propagated and thrown as an error in the calling environment *when the future value is requested*. For example, if we use lazy evaluation on a future that generates an error, we might see something like

```
> plan(sequential)
> b <- "hello"
> a %<-% {
+     cat("Resolving 'a' ...\n")
+     log(b)
+ } %lazy% TRUE
> cat("Everything is still ok although we have created a future that
Everything is still ok although we have created a future that will fa
> a
Resolving 'a' ...
Error in log(b) : non-numeric argument to mathematical function
```

The error is thrown each time the value is requested, that is, if we try to get the value again will generate the same error:

```
> a
Error in log(b) : non-numeric argument to mathematical function
In addition: Warning message:
restarting interrupted promise evaluation
```

To see the *last* call in the call stack that gave the error, we can use the `backtrace()` function(*) on the future, i.e.

```
> backtrace(a)
[[1]]
log(a)
```

(*) The commonly used `traceback()` does not provide relevant information in the context of futures. Furthermore, it is unfortunately not possible to see the list of calls (evaluated expressions) that led up to the error; only the call that gave the error (this is due to a limitation in `tryCatch()` used internally).

Globals

Whenever an R expression is to be evaluated asynchronously (in parallel) or sequentially via lazy evaluation, global (aka “free”) objects have to be identified and passed to the evaluator. They need to be passed exactly as they were at the time the future was created, because, for lazy evaluation, globals may otherwise change between when it is created and when it is resolved. For asynchronous processing, the reason globals need to be identified is so that they can be exported to the process that evaluates the future.

The future package tries to automate these tasks as far as possible. It does this with help of the globals package, which uses static-code inspection to identify global variables. If a global variable is identified, it is captured and made available to the evaluating process. Moreover, if a global is defined in a package, then that global is not exported. Instead, it is made sure that the corresponding package is attached when the future is evaluated. This not only better reflects the setup of the main R session, but it also minimizes the need for exporting globals, which saves not only memory but also time and bandwidth, especially when using remote compute nodes.

Finally, it should be clarified that identifying globals from static code inspection alone is a challenging problem. There will always be corner cases where automatic identification of globals fails so that either false globals are identified (less of a concern) or some of the true globals are missing (which will result in a run-time error or possibly the wrong results). Vignette 'Futures in R: Common Issues with Solutions' provides examples of common cases and explains how to avoid them as well as how to help the package to identify globals or to ignore falsely identified globals. If that

does not suffice, it is always possible to manually specify the global variables by their names (e.g. `globals = c("a", "slow_sum")`) or as name-value pairs (e.g. `globals = list(a = 42, slow_sum = my_sum)`).

Constraints when using Implicit Futures

There is one limitation with implicit futures that does not exist for explicit ones. Because an explicit future is just like any other object in R it can be assigned anywhere/to anything. For instance, we can create several of them in a loop and assign them to a list, e.g.

```
> plan(multiprocess)
> f <- list()
> for (ii in 1:3) {
+     f[[ii]] <- future({
+         Sys.getpid()
+     })
+ }
> v <- lapply(f, FUN = value)
> str(v)
List of 3
 $ : int 28602
 $ : int 28603
 $ : int 28605
```

This is *not* possible to do when using implicit futures. This is because the `%<-%` assignment operator *cannot* be used in all cases where the regular `<-` assignment operator can be used. It can only be used to assign future values to *environments* (including the calling environment) much like how `assign(name, value, envir)` works. However, we can assign implicit futures to environments using *named indices*, e.g.

```
> plan(multiprocess)
> v <- new.env()
> for (name in c("a", "b", "c")) {
+     v[[name]] %<-% {
+         Sys.getpid()
+     }
+ }
> v <- as.list(v)
> str(v)
List of 3
 $ a: int 28606
 $ b: int 28607
 $ c: int 28608
```

Here `as.list(v)` blocks until all futures in the environment `v` have been resolved. Then their values are collected and returned as a regular list.

If *numeric indices* are required, then *list environments* can be used. List environments, which are implemented by the listenv package, are regular environments with customized subsetting operators making it possible to index them much like how lists can be indexed. By using list environments where we otherwise would use lists, we can also assign implicit futures to list-like objects using numeric indices. For example,

```
> library("listenv")
> plan(multiprocess)
> v <- listenv()
> for (ii in 1:3) {
+     v[[ii]] %<-% {
+         Sys.getpid()
+     }
+ }
> v <- as.list(v)
> str(v)
List of 3
 $ : int 28609
 $ : int 28610
 $ : int 28611
```

As previously, `as.list(v)` blocks until all futures are resolved.

Demos

To see a live illustration how different types of futures are evaluated, run the Mandelbrot demo of this package. First, try with the sequential evaluation,

```
library("future")
plan(sequential)
demo("mandelbrot", package = "future", ask = FALSE)
```

which resembles how the script would run if futures were not used. Then, try multiprocess evaluation, which calculates the different Mandelbrot planes using parallel R processes running in the background. Try,

```
plan(multiprocess)
demo("mandelbrot", package = "future", ask = FALSE)
```

Finally, if you have access to multiple machines you can try to set up a cluster of workers and use them, e.g.

```
plan(cluster, workers = c("n2", "n5", "n6", "n6", "n9"))
demo("mandelbrot", package = "future", ask = FALSE)
```

Contributing

The goal of this package is to provide a standardized and unified API for using futures in R. What you are seeing right now is an early but sincere attempt to achieve this goal. If you have comments or ideas on how to improve the 'future' package, I would love to hear about them. The preferred way to get in touch is via the GitHub repository, where you also find the latest source code. I am also open to contributions and collaborations of any kind.