

Redis for Market Monitoring

Dirk Eddelbuettel¹

¹ Department of Statistics, University of Illinois, Urbana-Champaign, IL, USA

This version was compiled on March 15, 2022

This note shows how to use Redis cache (near-)real-time market data, and utilise its publish/subscribe (“pub/sub”) facility to distribute the data.

Overview

Redis (Sanfilippo, 2009) is a popular, powerful, and widely-used ‘in-memory database-structure store’ or server. We provide a brief introduction to it in a sibling vignette (Eddelbuettel, 2022) that is also included in package **RcppRedis** (Eddelbuettel and Lewis, 2022).

This note describes an interesting use case and illustrates both the ability of Redis to act as a (short-term) data cache (for which Redis is very frequently used) but also rely on its ability to act as “pub/sub” message broker. The “pub/sub” (short for “publish/subscribe”) framework is common to distribute data in a context where (possibly a large number of) “subscribers” consume data provided by one or a few services, often on a local network. Entire libraries and application frameworks such as **ZeroMQ** by Hintjens and Sustrik (2010) (and literally hundreds more) have pub/sub at its core. But as this note shows, one may not need anything apart from a (possibly already existing) Redis client.

Use Case: Market Data

Basics. Monitoring financial market data is a very common task, and many applications address it. In package **dang** we provide a function `intradayMarketMonitor()` which extends earlier work by Ulrich (2021) and does just that for the SP500 index and its symbol `^GSPC` (at Yahoo! Finance). For non-tradeable index symbols such as `^GSPC` one can retrieve near-“real-time” updates which is nice. We put “real-time” in quotes here as there are of course delays in the transmission from the exchange or index provide to a service such as Yahoo! and then down a retail broadband line to a consumer. Yet it is “close” to real-time—as opposed to explicitly delayed data that we cover below. So `intradayMarketMonitor()` runs in an endless loop, updates the symbol and plot, and after market close once writes its history into an RDS file so that a restart can access some history. It is nicely minimal and self-contained design.

Figure 1 shows a plot resulting from calling the function on a symbol, here again `^GSPC`, when two days of history have been accumulated. (The plot was generated on a weekend with the preceding Friday close providing the last data point.)

Possible Shortcomings. Some of the short-comings of the approach in `intradayMarketMonitor()` and Ulrich (2021) are

- use of one R process per symbol
- same process used for monitoring and plotting
- no persistence until end of day

Moreover, the ‘real-time’ symbol for the main market index is available only during (New York Stock Exchange) market hours. Yet sometimes one wants to gauge a market reaction or ‘mood’ at off-market hours.

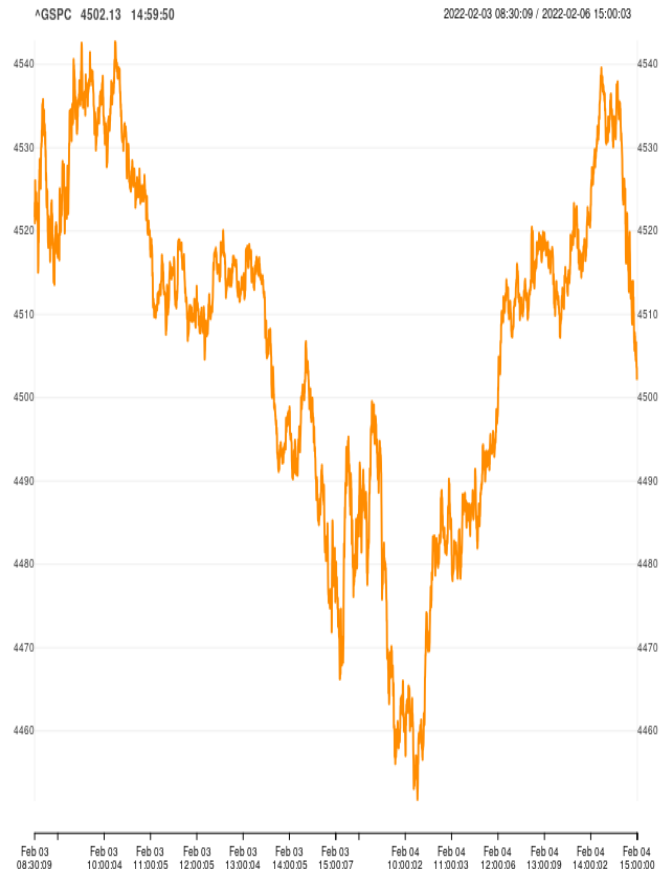


Fig. 1. Intraday Market Monitoring Example

So with this, the idea arose to decouple market data *acquisition and caching* from actual *visualization* or other monitoring. This would also permit distributing the tasks over several machines: for example an ‘always-on’ monitoring machine could always track the data and store it for other ‘on-demand’ machines or applications to access it. And as we have seen, Redis makes for a fine data ‘caching’ mechanism.

Building A Market Monitor

Data. The **quantmod** package by Ryan and Ulrich (2020a) provides a function `getQuote()` we can use to obtain data snapshots. We will look at `^GSPC` as before but also `ES=F`, the Yahoo! Finance symbol for the ‘rolling front contract’ for the SP500 Futures trading at CME Globex under symbol `ES`. (We will not get into details on futures contracts here as the topic is extensively covered elsewhere. We will just add that equity futures tend to trade in only one contract (“no curve”) and roll to the next quarterly expiration at particular dates well established and known by market practice.)

```
suppressMessages(library(quantmod))
res <- getQuote(c("^GSPC", "ES=F", "SPY"))
res[,1:3] # omitting chg, OHL, Vol
#           Trade Time      Last      Change
# ^GSPC 2022-02-11 17:05:08 4418.64 -85.43994
# ES=F 2022-02-11 16:59:59 4418.00 -79.50000
# SPY 2022-02-11 16:00:01 440.46 -8.86002
```

The preceding code display shows how the **quantmod** (Ryan and Ulrich, 2020a) function `getQuote()` can access index data (symbol '^GSPC'), futures data (symbol 'ES=F' as the rolling front contract) as well as equity / ETF data (symbol 'SPY').

Storing and Publishing. Given per-security rows of data as shown in the preceding example, we can use **Redis** to store the data using the timestamp as a sorting criterion in a per-symbol stack. The 'sorted set' data structure is very appropriate for this. The function `get_data()` transforms the result of `getQuote()` into a named numeric vector suitable for our use of 'sorted sets'.

```
get_data <- function(symbol) {
  quote <- getQuote(symbol)
  vec <- c(Time = as.numeric(quote$`Trade Time`),
          Close = quote$Last,
          Change = quote$Change,
          PctChange = quote$`% Change`,
          Volume = quote$Volume)
  vec
}
```

Similarly, given the symbol, we can also 'publish' a datum with the current values and timestamp. In the example application included with **Redis**, this is done by relying on the following short function which receives the current data record and then stores and publish it.

```
store_data <- function(vec, symbol) {
  redis$zadd(symbol, matrix(vec, 1))
  redis$publish(symbol, paste(vec, collapse=";"))
}
```

In this example, the **redis** instance is a script-level global symbol. This could easily be rewritten where it is also be passed into the function, and `vec` is a simple vector of observations procured by `getQuote()` as discussed in the preceding code example. The timestamp is transformed into a numeric value making the vector all-numeric which the format used by `zadd()` to added a 'sorted' (by the timestamp) numeric one-row matrix. Beside storing the data, we also publish it via **Redis** on channel named as the symbol. Here the numeric data is simply concatenated with a ; as separator and sent as text.

The core functionality in the main loop is then as follows below where we also omitted some of the error or status messaging for brevity.

In that example, the change is volume is used as a 'tell' for actual new data. This works reliably for the (main futures) markets we follow here which have essentially constant trading activity. When some tranquil periods occur, the gaps between stored and published data points may be longer than the default sleep period of ten seconds used here.

```
y <- try(get_data(symbol), silent = TRUE)
if (inherits(y, "try-error")) {
  msg(curr_t, "Error ...") # rest omitted
  errored <- TRUE
  Sys.sleep(15)
  next
} else if (errored) {
  errored <- FALSE
  msg(curr_t, "...recovered")
}
v <- y["Volume"]
if (v != prevVol) {
  store_data(y, symbol)
  msg(curr_t, "Storing ...") # same
}
prevVol <- v
Sys.sleep(10)
```

The remainder of the 'acquiring data and storing in **Redis**' code is similar to the non-**Redis** using variant `intradayMarketMonitor()` in **dang** (Eddelbuettel, 2021) that is based on the earlier work by **Ulrich** (2021).

Retrieving and Subscribing. Two core routines to receive data from **Redis** to plot both read the most recent stored data at startup, and then grow this data set via a subscription to the updates published to the channel.

We first show the initial request for all data, which is then subset to the n most recent days. We can request 'all' data as we also deploy a helper script referenced in the appendix to keep the overall data volume that is stored at 'manageable' and finite levels. Adding such a step is important for a process such as this which continually appends data which, if unchecked, would 'eventually' exhaust system resources.

```
most_recent_n_days <- function(x, n=2,
                              minobs=1500) {
  tt <- table(as.Date(index(x)))
  if (length(tt) < n) return(x)
  ht <- head(tail(tt[tt>minobs], n), 1)
  cutoff <- paste(format(as.Date(names(ht))),
                  "00:00:00")
  newx <- x[ index(x) >= as.POSIXct(cutoff) ]
  msg(Sys.time(), "most recent data starting at",
      format(head(index(newx), 1)))
  newx
}

get_all_data <- function(symbol, host) {
  m <- redis$zrange(symbol, 0, -1)
  colnames(m) <- c("Time", "Close", "Change",
                  "PctChange", "Volume")
  y <- xts(m[, -1],
          order.by=anytime(as.numeric(m[, 1])))
  y
}

## ... some setup
x <- get_all_data(symbol, host)
x <- most_recent_n_days(x, ndays)
```

The updates from subscription happen in the main while() loop. The subscription is set up as follows:

```
## This is the callback func. assigned to a symbol
.data2xts <- function(x) {
  m <- read.csv(text=x, sep=";", header=FALSE,
               col.names=c("Time", "Close",
                           "Change", "PctChange",
                           "Volume"))
  y <- xts(m[, -1, drop=FALSE],
          anytime(as.numeric(m[, 1, drop=FALSE])))
  y
}
# programmatic version of `ES=F` <- function(x) ...
assign(symbol, .data2xts)
redis$subscribe(symbol)
```

The .data2xts() callback function parses the concatenated values, and constructs a one-row object xts object. The xts package by Ryan and Ulrich (2020b) make time-ordered appending of such data via rbind easy which is what is done in the main loop:

```
y <- redisMonitorChannels(redis)
if (!is.null(y)) {
  x <- rbind(x,y)
  x <- x[!duplicated(index(x))]
}
show_plot(symbol, x)
```

The redisMonitorChannels(redis) is key to our pub/sub mechanism here. The subscriptions are stored in the redis instance, along with any optional callbacks. The function will listen to (one or more) channels using the key Redis function listen() and consume the next message. The key here is our addition of an optional per-symbol callback which, if present, is used to process the returned data. This means that in our application with the .data2xts() function used as a per-symbol callback, the returned variable y above is a standard xts object which rbind efficiently appends to an existing object which is how we grow x here. (For brevity we have omitted two statements messaging data upgrade process to the console when running, they are included in the full source file included in the package.)

Extending to Multiple Symbol

The pub/sub mechanism is very powerful. Listening to a market symbol, storing it, and publishing for use on local network enables and facilitates further use of the data.

Naturally, the idea arises to listen to multiple symbols. At first glance, one could run one listener process by symbol. The advantage is the ease of use. A clear disadvantage is the inefficient resource utilization.

And it turns out that we do not have to. Just how the initial quantmod::getQuote() call shows access to several symbols at once, we can then process a reply from getQuote() and store and publish multiple symbols on multiple channels. This is done in files intraday-GLOBEX-to-Redis.r and intraday-GLOBEX-from-Redis.r. Just like the initial examples for ES, these files show how to cover several symbols. Here we use for: Bitcoin, SP500, Gold, and WTI Crude Oil. By sticking to the same exchanges, here CME Globex, we can use one set of 'open' or 'close' rules.

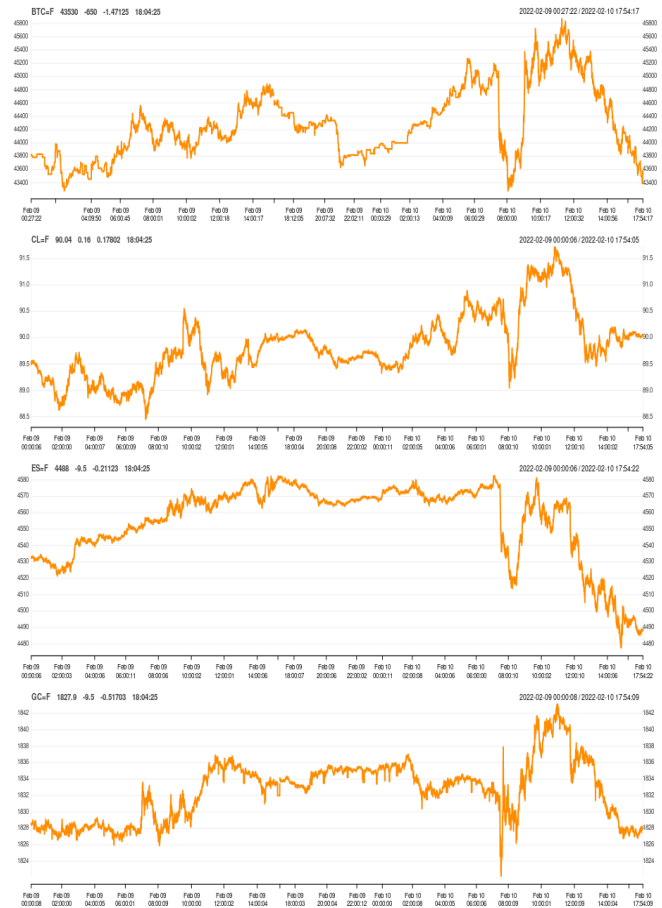


Fig. 2. Multi-Symbol Market Monitoring Example

Data and Publishing. The following snippet fetches the data and stores and publishes it.

```
symbols <- c("BTC=F", "CL=F", "ES=F", "GC=F")

get_data <- function(symbols) {
  quotes <- getQuote(symbols)
  quotes$Open <- quotes$High <- quotes$Low <- NULL
  colnames(quotes) <- c("Time", "Close", "Change",
                       "PctChange", "Volume")
  quotes$Time <- as.numeric(quotes$Time)
  quotes
}

store_data <- function(res) {
  symbols <- rownames(res)
  res <- as.matrix(res)
  for (symbol in symbols) {
    vec <- res[symbol, , drop=FALSE]
    redis$add(symbol, vec)
    redis$publish(symbol,
                  paste(vec, collapse=";"))
  }
}
```

It is used in the main loop inside a try() statement and error handler.

```

res <- try(get_data(symbols), silent = TRUE)
if (inherits(res, "try-error")) {
  msg(curr_t, "Error:",
        attr(res, "condition")[["message"]])
  errored <- TRUE
  Sys.sleep(15)
  next
} else if (errored) {
  errored <- FALSE
  msg(curr_t, "...recovered")
}
v <- res[3, "Volume"]
if (v != prevVol) {
  store_data(res)
  # msg(...omitted for brevity...)
}
prevVol <- v
Sys.sleep(10)

```

Retrieving data. The receiving side of the application works similarly. First, we need to subscribe to multiple channels:

```

env <- new.env() # local environment for callbacks

## same .data2xts() function as above

## With environment 'env', assign callback
## function for each symbol
res <- sapply(symbols, function(symbol) {
  ## progr. version of `ES=F` <- function(x) ...
  assign(symbol, .data2xts, envir=env)
  redis$subscribe(symbol)
})

```

We then use a slightly generalized listener:

```

## Callback handler for convenience
multiSymbolRedisMonitorChannels <-
  function(context,
           type="rdata", env=.GlobalEnv) {
    res <- context$listen(type)
    if (length(res) != 3 ||
        res[[1]] != "message") return(res)
    if (exists(res[[2]], mode="function",
              envir=env)) {
      data <- do.call(res[[2]],
                    as.list(res[[3]]),
                    envir=env)
      val <- list(symbol=res[[2]],
                 data=data)
      return(val)
    }
    res
  }

```

The `listen` methods returns an object which is checked for correct length and first component. If appropriate, the second element is the channel symbol so if a callback function of the same names exists, it is called with the third element, the ‘payload’. This creates the familiar `xts` object with is return along with the symbol in a two-element list.

The data is consumed in the while loop in a very similar fashion to the one-symbol case, but we now unpack the loop and operate on the appropriate data element.

```

## monitor channels in context of 'env'
rl <- multiSymbolRedisMonitorChannels(redis,
                                     env=env)

if (is.list(rl)) {
  sym <- rl[["symbol"]]
  x[[sym]] <- rbind(x[[sym]], rl[["data"]])
  z <- tail(x[[sym]],1)
  if (sym == symbols[3]) msg(#...omitted...)
} else {
  msg(index(now_t), "null data in y")
}
show_plot(symbols, x)

```

Finally, the plot function simply plots for all symbols in the `symbols` vector.

Overall, this setup is robust to data ‘surprises’ as the `try()` mechanism implements an error recovery in cases of temporary network or remote server issues. The overall design is simple: each of the two files for, respectively, receiving-and-storing data and accessing-and-visualizing, contains only a few short helper functions (most of which where shown above) and a core `while()` loop. We have had these running uninterrupted and without issues for months on end.

Summary

We describe a simple yet efficient mechanism to capture and publish ‘live’ market data by relying on Redis via the **RcppRedis** package.

Acknowledgements

Joshua Ulrich provided a first useable monitoring loop for a life symbol which is gratefully acknowledged, as are numerous discussions about **quantmod** and other packages. Bryan Lewis not only put an elegant and working pub/sub mechanism in his **redis**, but also ported it into a very elegant callback-based solution in package **RcppRedis**. These features, and this monitoring application, would not exist without the help of either Josh or Bryan.

Appendix

Data Growth. The scripts do not write the data to Redis with a ‘time-to-live’ (TTL) expiry. This means the database is growing. A simple way to limit the growth is to invoke a pruning script from cron once a week. We include a simple script in the `pub-sub/` directory of the package.

References

- Eddelbuettel D (2021). *dang: 'Dang' Associated New Goodies*. R package version 0.0.15, URL <https://CRAN.R-project.org/package=dang>.
- Eddelbuettel D (2022). “A Brief Introduction to Redis.” doi: [10.48550/arXiv.2203.06559](https://doi.org/10.48550/arXiv.2203.06559).
- Eddelbuettel D, Lewis BW (2022). *RcppRedis: 'Rcpp' Bindings for 'Redis' using the 'hiredis' Library*. R package version 0.2.0, URL <https://CRAN.R-project.org/package=RcppRedis>.
- Hintjens P, Sustrik M (2010). “ZeroMQ: An open-source universal messaging library.” <https://zeromq.org>.
- Ryan JA, Ulrich JM (2020a). *quantmod: Quantitative Financial Modelling Framework*. R package version 0.4.18, URL <https://CRAN.R-project.org/package=quantmod>.

Ryan JA, Ulrich JM (2020b). *xts: eXtensible Time Series*. R package version 0.12.1, URL <https://CRAN.R-project.org/package=xts>.
Sanfilippo S (2009). "Redis In-memory Data Structure Server." <https://redis.io>.
Ulrich JM (2021). "Market-Monitoring with R." <https://gist.github.com/joshualrich/ee11ef67b1461df399b84efd3c8f9f67#file-intraday-sp500-r>.