Redis for Market Monitoring

Dirk Eddelbuettel[1]

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

This version was compiled on February 10, 2022

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

Overview

[Redis][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 sibbling vignette (Eddelbuettel, 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][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 (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][redis] client.

Use Case: Market Data

Basics. Monitoring financial market data is a very common application. In package **dang** we provide a function `intradayMarketMonitor()` (based on and extending an earlier version by Josh Ulrich published in this gist) which 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 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()` 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.

So with this, 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][redis] makes for a fine data 'caching' mechanism.

Fig. 1. Intraday Market Monitoring Example

Building A Market Monitor

Data. The **quantmod** package 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"))
res[,1:3] # omitting chg, OHL, Vol
#                Trade Time     Last   Change
#  ^GSPC 2022-02-10 17:05:48 4504.08 -83.1001
#  ES=F  2022-02-10 19:43:16 4477.75 -19.7500
```

Storing and Publishing. Given such a per-security row of data, we can use [Redis][redis] to store the data (given the timestamp as a sorting criterion) in a per-symbol stack. The Sorted Set data structure is very appropriate for this. Similarly, given the symbol we can publish a datum with the current values and timestamp. In the example application included with **Redis**, this is done by the following function:

```
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
}

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

Here the `redis` instance is global, it could also be passed into the function. `vec` is simply vector of observation procured by `getQuote()` as discussed in the preceding code exampled. 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 remainder of the 'acquiring data and storing in Redis' code is similar to the non-Redis varian `intradayMarketMonitor()` in **dang** Eddelbuettel (2021).

Retrieving and Subscribing. The sibbling routine to receive data from Redis to plot both reads recent stored data once at startup, and then grows the this data set via a subscription to the updates published to the channel.

We first show the initial request of all data, which is then subset to the *n* most recent days:

```
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 (2020) 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 the pub/sub mechanism. Subscriptions are stored in the `redis` instance, along with any optional callbacks. The function will listen to (one or more) channels and consume the next message, which it returns processed via the callbacl. This means that variable `y` above is 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 file.)

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.

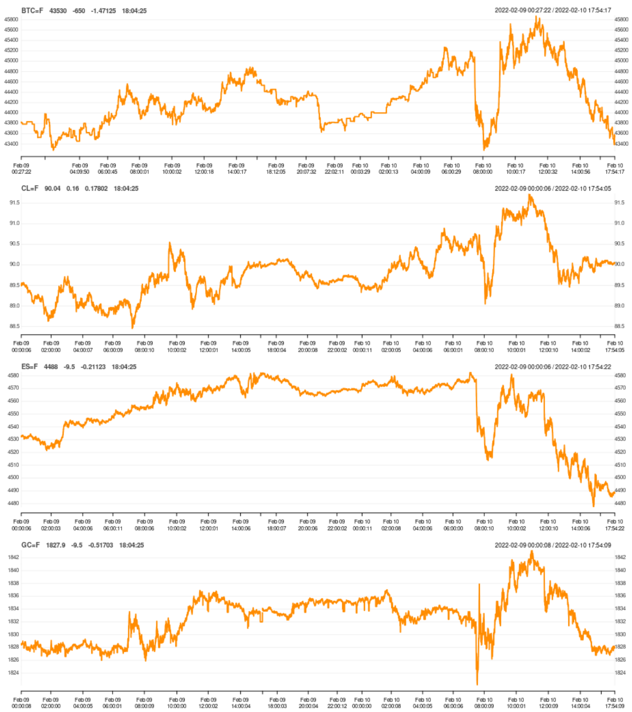

Fig. 2. Multi-Symbol Market Monitoring Example

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$zadd(symbol, vec)
        redis$publish(symbol,
                      paste(vec,collapse=";"))
    }
}
```

It is use in the main loop in 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. The receiving side works similarly. 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 slighly 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. [redis]: https://redis.io [hiredis]: https://github.com/redis/hiredis

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." Vignette in package **RcppRedis**. URL https://CRAN.R-Project.org/package=RcppRedis.

Ryan JA, Ulrich JM (2020). *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.