Real Time Market Data and Trade Execution with R

Jeffrey A. Ryan

April 5, 2009

Contents

Abstract

While many trading systems in production make use of different languages for different tasks, the ideal environment involves a single language that can handle incoming data, make trade decisions, and execute all from one environment. This paper explores the issues relating to managing this entire process in the R programming langauge.

1 Introduction to Real Time Processing in R

R [1] is a statistical programming language based on the S [?] dialect created by Chambers et al at Bell Laboratories.

As a scripting language with a well-developed interpreter it is ideally suited for data exploration as well as rapid programming. More information as to the merits of R for modeling and development can be found in countless papers and books. It is assumed that the reader has already been exposed to the virtues of R, and this paper will focus on extending the prototypical use to managing real-time processing.

While the R language has traditionally been used for data analysis and exploration, as well as pre-trade and post-trade modeling, it can also be used successfully as a platform to manage the entire trade process. This paper will explore the rationale, design, implementation, and limitations of using R for managing algorithmic trading with R. It will conclude with an overview of the open source IBrokers package that interfaces the Interactive Brokers brokerage platform from R.

1.1 Socket Connections in R

R provides a small collection of tools to manage socket based connections from within the interactive environment. While most cases involve the opening of a connection, some read or write operation, and a subsequent close, it is necessary to extend this logic to persistent connections to accommodate market data and asynchonous incoming and outgoing messages.

1.1.1 Overview

The basic tools in R to manage socket-based connections involve five primary functions, with a susbset of interest including three functions. These are `socketConnection`, `writeBin`, and `readBin`.

The establish a connection the socketConnection function is called. This call returns an R connection object. This connection can be used to send and receive data from sockets. Connection types can be specified using the *open=* argument. Additional details can be found in the respective help pages.

Connections can be either text-based, or binary based, depending on the open argument. The functions used with text connection objects returned by socketConnect are *readChar* and *writeChar*. For network connections *readBin* and *writeBin* are often more appropriate. Byte-ordering is handled by these functions as well.

1.1.2 Usage

Cover connecting with socketConnection, the read/write semantics, as well as cross-platform differences. Provide general insight into the process as well as practical consideration with respect to incoming message construction/headers.

1.2 Technical Challenges in `R`

While the idea of using R to process incoming data, apply trade logic, and place orders makes inuitive sense, it is not without technical challenges. These range from the single-threaded nature of R, to the fact that R is interpreted and thus relatively slower than a compiled solution. These challenges will be discussed in the following sections.

1.2.1 Asynchronous Requests

As R is currently single-threaded, and for the foreseable future will remain so, this becomes a major hurdle to implementing an algorithmic solution within R.

The conventional wisdom is that managing asynchronous events within a single thread is difficult at best, or impossible at worst. As with most things, the true answer lies somewhere in the middle.

Most trade decisions are single-threaded by design: e.g. a price target is met and an order is placed. Even trade logic that is not so simple can usually be factored out to something that fits within this action|reaction paradigm.

The key challenge in managing this process within a single process is to be able to manage incoming data external to the trade decision, and have ready access to it when needed.

The limitation of a single-threaded model is that all incoming messages must be able to be handled within the processing constraints of a single R thread. The primary constraint in this process is the need to loop through each distinct message, as each message involves a seperate call or calls to the underlying `readBin`.

The structure of the incoming message feed, as well as the computational requirements to parse the feed are integral to the upper limit that this solution can provide.

1.2.2 Socket Connection Limits

As alluded to above, the primary limitation of managing an event stream in R is that without threads to absord additional messages, all incoming data must be manageable within one process and one main event loop.

This limitation is most severe with high frequency data, specifically streaming market data or limit-order book data. Depending on the structure of the incoming data, each distinct message must be dealt with during one pass through the event loop described below. If messages are grouped into aggregate structures containing many distinct messages internally only one call to `readBin` will be required, and throughput can be improved dramatically. This is dependent on the incoming feed, either from the provider, or an intermediary process.

1.2.3 Event Loops

The typical way to manage real-time event streams in any language would be to take each message as a distinct unit, process any header information that

is contained, and read the remaining data into memory. From this point the individual logic would be application dependent.

Event streams in R follow this same format, where an infinite loop is constructed using a `while` loop, and at each iteration a new message is read from the connection. This message is then passed to a processing function that branches based on the message type. Further handling is application specific.

1.2.4 Data Persistence and Sharing

The final piece to the puzzle for managing live data in R is the ability to maintain state information. Within the constraints of a single R process and event loop it is imperative to maintain a list of data representing current and possibly recently historical market prices and sizes, as well as a position type information.

One relatively elegant solution in R is to use environments and closures to be able to manage arbitrary data structures from any part of a process. Data managed in this way is accessible quickly and efficiently from within event handlers.

2 IBrokers: Using the Interactive Brokers API

To illustrate the process of using R as an algorithmic trading environment we will examine how the `IBrokers` package manages the process.

2.1 Package Overview

Interactive Brokers is an electronic brokerage platform that is popular among active and algorithmic traders. They provide a platform for individual and institutional clients, as well as a full API to execute orders from a variety of platforms.

The API is of particular interest for algorithmic trading. A supported client toolkit to access this API is provided in Java, C++, and Excel, using sockets and Microsoft DDE infrastructure. Numerous additional interfaces have been built by the community for programming languages ranging from Python and Perl, to Matlab and C. A previously unsupported platform had been R. The IBrokers package by this author is available for no charge from CRAN. This package provides a direct interface to the API via socket connections implemented entirely in R.

In this section we will explore the design objectives of the IBrokers package. We will also examine the API implementation details within the context of using R for managing real-time data.

2.2 Design Criteria

The primary design requirement of IBrokers was to allow for rapid prototyping of trading ideas from within R, as well as implementation of R-based trade logic, using Interactive Brokers as both a data feed and execution platform. This has

subsequently evolved to include the possibility of incorporating a separate data feed external to Interactive Brokers for the purposes of flexibility and robustness. This last point is currently experimental and only supported on non-Windows platforms, where R's underlying readBin function is well behaved.

2.3 Message Structure

The basic structure of the formal API allows for asynchronous outgoing and incoming messages. The internal design of incoming messages from the Trader-Workstation (TWS) is nul terminated byte (char) arrays. These are easily managed natively in R by `readBin` using arguments `what=character()` or `what=raw()` to handle the incoming data types. The IBrokers implementation makes use of processing events as 8-bit char arrays as handled by `what=character()`, which automatically reads to the first nul-byte and assigns to a character vector.

Each incoming message consists of a single two or three byte char array as a header, representing a numeric value in the range of 1 to 50, followed by an ASCII nul-byte The remainder of the message is comprised of a variable number of nul-terminated messages, with the length dependent on the incoming message type. The full set of incoming message types can be seen in the internal IBrokers variable `.twsIncomingMSG`.

Outgoing messages follow the same general logic, with a header code, followed by the required fields of the request. These codes can be seen in the internal variable `.twsOutgoingMSG`.

2.4 Implementation

The basic logic of the `IBrokers` implementation is comprised of four main components:

1. a request message or messages [e.g. embedded in `reqMktData`]
2. a primary callback function [`twsCALLBACK`]
3. a message/event loop within the callback [`processMsg`]
4. a data/method closure [`eWrapper`] to process individual messages and facilitate trade logic.

2.4.1 Requests

The basic TWS request is for a single return reply or, as is the case for data requests, a subscription that will remain active until an appropriate cancel request is made. Using IBrokers built-in requests, most all subscription requests will be automatically cancelled upon function error or exit. User generated requests that utilize the raw API (bypassing IBrokers wrapper functions) will need to request cancels where appropriate.

The basic request as outlined above consists of a headed code sent as a character array (a length one character vector in R). The precise structure of

the message is defined by the API and any deviation will result in an error at some point in the subsequent process.

For example, to request the current time from the TWS, one needs to send the code for "Current Time" (.twsOutgoingMSG$REQ_CURRENT_TIME): "49" and the current version number of the specific request. In the case of current time, the version is simply the character "1".

```
> writeBin(c("49", "1"), con)
```

2.4.2 Callback

The callback within IBrokers is in fact the first level of callbacks that are used to manage the events sent and returned from the TWS.

After a request has been sent to the TWS, the function passed via the `CALLBACK` argument is called. By default this is the main `twsCALLBACK` wrapper. The basic control structure is that within this callback an event handler object is created, and subsequently passed into an infinite event loop. Within this loop new messages are read and in turn passed into the `processMsg` function for individual handling. A common object that is used at each level of the process is the event handler. This is where customized data management logic can be added, as well as data stored and read from.

In addition to the event handler being available to user level customization, the main event loop itself (the one calling processMsg on each new message) should contain code that implements the trading logic to be employed in a given strategy or strategies.

Examining the relevant section of twsCALLBACK we can see exactly where this logic can be inserted:

```
> require(IBrokers)

IBrokers version 0.2-8: (alpha)
Implementing API Version 9.63

This software comes with NO WARRANTY.  Not intended for production use!
See ?IBrokers for details

> deparse(twsCALLBACK)[30:47]

 [1] "                  Sys.sleep(5 * playback)"
 [2] "            }"
 [3] "        }"
 [4] "    }"
 [5] "    else {"
 [6] "        while (TRUE) {"
 [7] "            socketSelect(list(con), FALSE, NULL)"
 [8] "            curMsg <- .Internal(readBin(con, \"character\", 1L, "
 [9] "                NA_integer_, TRUE, FALSE))"
```

```
[10] "            if (!is.null(timestamp)) {"
[11] "                processMsg(curMsg, con, eWrapper, format(Sys.time(), "
[12] "                  timestamp), file, ...)"
[13] "            }"
[14] "            else {"
[15] "                processMsg(curMsg, con, eWrapper, timestamp, "
[16] "                  file, ...)"
[17] "            }"
[18] "        }"
```

At line #2 the call to readBin returns the current message from the TWS, here stored as `curMsg`. This will be any of the fifty or so incoming message types as defined by the API.

A check to make sure that the message isn't empty is then made. As we may be employing non-blocking connections (true by default), we choose to add a system delay up to 1/100th of a second to ease the computational overhead of the request loop. This isn't necessary and will be irrelevant for blocking connections, but is good practice to maintain.

Assuming the second if-else branch is taken, the code now dispatches to the processMsg call, which itself is simply a large if-else construct with a single call to the apporpriate functional method set by the `eWrapper` object (closure).

The processMsg event handler returns nothing and is only used to create side-effects. Most useful of these are to update the environment that is attached to the `eWrapper` object with new data.

Following this processing of the current message, it would be possible to add trade logic based on the data contained in the eWrapper environment. An implementation of this will be clear in the example that follows this section.

(a closure created in R, containing an environment for data, as well as a list of functions to handle all possible incoming messages from the TWS)

2.4.3 Event Loop

2.4.4 Closure

2.5 Example: A Software Implemented Stop Order with IBrokers

<SHOW SCHEMATIC HERE>

References

[1] R Development Core Team: *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, URL http://www.R-project.org