Distributed Computing Patterns in R

Whit Armstrong
armstrong.whit@gmail.com

KLS Diversified Asset Management

May 17, 2013

Messaging patterns

- Messaging patterns are ways of combining sockets to communicate effectively.
- In a messaging pattern each socket has a defined role and fulfills the responsibilities of that role.
- ZMQ offers several built-in messaging patterns which make it easy to rapidly design a distributed application:
  - Request-reply, which connects a set of clients to a set of services.
  - Pub-sub, which connects a set of publishers to a set of subscribers.
  - Pipeline, which connects nodes in a fan-out/fan-in pattern that can have multiple steps and loops.
  - Exclusive pair, which connects two sockets exclusively.

What does ZMQ give us?

- ZMQ is a highly specialized networking toolkit.
- It implements the basics of socket communications while letting the user focus on the application.
- Very complex messaging patterns can be built on top of these simple ZMQ sockets (Paranoid Pirate, Majordomo, Binary Star, Suicidal Snail, etc.).
- I highly recommend reading “The Guide” before writing your own apps.
- `http://zguide.zeromq.org/page:all`

Request / Reply example

- Req / Rep is the most basic message pattern.
- Both the request socket and reply socket are synchronous.
- The reply socket can only service one request at a time, however, many clients may connect to it and queue requests.

Request / Reply, Server

```r
require(rzmq)

ctx <- init.context()
responder <- init.socket(ctx, "ZMQ_REP")
bind.socket(responder, "tcp://*:5555")

while (1) {
    req <- receive.socket(responder)
    send.socket(responder, "World")
}
```

Request / Reply, Client

```
require(rzmq)

requester <- init.socket(ctx, "ZMQ_REQ")
connect.socket(requester, "tcp://localhost:5555")

for (request.number in 1:5) {
    print(paste("Sending Hello", request.number))
    send.socket(requester, "Hello")
    reply <- receive.socket(requester)
    print(paste("Received:", reply, "number", request.number))
}

## [1] "Sending Hello 1"
## [1] "Received: World number 1"
## [1] "Sending Hello 2"
## [1] "Received: World number 2"
## [1] "Sending Hello 3"
## [1] "Received: World number 3"
## [1] "Sending Hello 4"
## [1] "Received: World number 4"
## [1] "Sending Hello 5"
## [1] "Received: World number 5"
```

Request / Reply server as remote procedure call

```r
require(rzmq)

ctx <- init.context()
responder <- init.socket(ctx, "ZMQ_REP")
bind.socket(responder, "tcp://*:5557")

while (1) {
    req <- receive.socket(responder)
    send.socket(responder, req * req)
}
```

Request / Reply client as remote procedure call

```r
require(rzmq)

requester <- init.socket(ctx, "ZMQ_REQ")
connect.socket(requester, "tcp://localhost:5557")

x <- 10
send.socket(requester, x)
reply <- receive.socket(requester)
all.equal(x * x, reply)

## [1] TRUE

print(reply)

## [1] 100
```

Request / Reply client – rpc server with user function

```r
require(rzmq)

ctx <- init.context()
responder <- init.socket(ctx, "ZMQ_REP")
bind.socket(responder, "tcp://*:5558")

while (1) {
    msg <- receive.socket(responder)
    fun <- msg$fun
    args <- msg$args
    result <- do.call(fun, args)
    send.socket(responder, result)
}
```

Request / Reply client – rpc client with user function

```
require(rzmq)

requester <- init.socket(ctx, "ZMQ_REQ")
connect.socket(requester, "tcp://localhost:5558")

foo <- function(x) {
    x * pi
}
req <- list(fun = foo, args = list(x = 100))
send.socket(requester, req)
reply <- receive.socket(requester)
print(reply)

## [1] 314.2
```

Realistic example – c++ server

```cpp
#include <string>
#include <iostream>
#include <stdexcept>
#include <unistd.h>
#include <zmq.hpp>
#include <boost/date_time/posix_time/posix_time.hpp>
#include <order.pb.h>
#include <fill.pb.h>
using namespace boost::posix_time;
using std::cout; using std::endl;

int main () {
  zmq::context_t context(1);
  zmq::socket_t socket (context, ZMQ_REP);
  socket.bind ("tcp://*:5559");

  while (true) {
    //  wait for order
    zmq::message_t request;
    socket.recv(&request);

    tutorial::Order o;
    o.ParseFromArray(request.data(), request.size());

    std::string symbol(o.symbol());
    double price(o.price());
    int size(o.size());

    // send fill to client
    tutorial::Fill f;
    f.set_timestamp(to_simple_string(microsec_clock::universal_time()));
    f.set_symbol(symbol); f.set_price(price); f.set_size(size);

    zmq::message_t reply (f.ByteSize());
    if(!f.SerializeToArray(reply.data(),reply.size())) {
      throw std::logic_error("unable to SerializeToArray.");
    }
    socket.send(reply);
  }
  return 0;
}
```

Realistic example – R client

```r
broker <- init.socket(ctx, "ZMQ_REQ")
connect.socket(broker, "tcp://*:5559")

## read the proto file
readProtoFiles(files = c("code/proto.example/order.proto", "code/proto.example/fill.proto"))

aapl.order <- new(tutorial.Order, symbol = "AAPL", price = 420.5, size = 100L)
aapl.bytes <- serialize(aapl.order, NULL)

## send order
send.socket(broker, aapl.bytes, serialize = FALSE)
## pull back fill information
aapl.fill.bytes <- receive.socket(broker, unserialize = FALSE)
aapl.fill <- tutorial.Fill$read(aapl.fill.bytes)
writeLines(as.character(aapl.fill))

## timestamp: "2013-May-16 17:33:41.619589"
## symbol: "AAPL"
## price: 420.5
## size: 100


esgr.order <- new(tutorial.Order, symbol = "ESGR", price = 130.9, size = 1000L)
esgr.bytes <- serialize(esgr.order, NULL)

## send order
send.socket(broker, esgr.bytes, serialize = FALSE)
## pull back fill information
esgr.fill.bytes <- receive.socket(broker, unserialize = FALSE)
esgr.fill <- tutorial.Fill$read(esgr.fill.bytes)
writeLines(as.character(esgr.fill))

## timestamp: "2013-May-16 17:33:41.627151"
## symbol: "ESGR"
## price: 130.9
## size: 1000
```

Pub / Sub example

- Pub / Sub is a more interesting pattern.
- The Pub socket is asynchronous, but the sub socket is synchronous.

Pub / Sub, Server

```
require(rzmq)

context = init.context()
pub.socket = init.socket(context, "ZMQ_PUB")
bind.socket(pub.socket, "tcp://*:5556")

node.names <- c("2yr", "5yr", "10yr")
usd.base.curve <- structure(rep(2, length(node.names)), names = node.names)
eur.base.curve <- structure(rep(1, length(node.names)), names = node.names)

while (1) {
    ## updates to USD swaps
    new.usd.curve <- usd.base.curve + rnorm(length(usd.base.curve))/100
    send.raw.string(pub.socket, "USD-SWAPS", send.more = TRUE)
    send.socket(pub.socket, new.usd.curve)

    ## updates to EUR swaps
    new.eur.curve <- eur.base.curve + rnorm(length(eur.base.curve))/100
    send.raw.string(pub.socket, "EUR-SWAPS", send.more = TRUE)
    send.socket(pub.socket, new.eur.curve)
}
```

Pub / Sub, USD Client

```r
require(rzmq)

subscriber = init.socket(ctx, "ZMQ_SUB")
connect.socket(subscriber, "tcp://localhost:5556")
topic <- "USD-SWAPS"
subscribe(subscriber, topic)

i <- 0
while (i < 5) {
    ## throw away the topic msg
    res.topic <- receive.string(subscriber)
    if (get.rcvmore(subscriber)) {
        res <- receive.socket(subscriber)
        print(res)
    }
    i <- i + 1
}

##   2yr   5yr  10yr
## 1.989 1.996 1.992
##   2yr   5yr  10yr
## 2.006 2.005 1.996
##   2yr   5yr  10yr
## 2.001 1.992 2.003
##   2yr   5yr  10yr
## 2.005 1.997 1.998
##   2yr   5yr  10yr
## 1.998 2.010 2.006
```

Pub / Sub, EUR Client

```
require(rzmq)

subscriber = init.socket(ctx, "ZMQ_SUB")
connect.socket(subscriber, "tcp://localhost:5556")
topic <- "EUR-SWAPS"
subscribe(subscriber, topic)
i <- 0
while (i < 5) {
    ## throw away the topic msg
    res.topic <- receive.string(subscriber)
    if (get.rcvmore(subscriber)) {
        res <- receive.socket(subscriber)
        print(res)
    }
    i <- i + 1
}

##    2yr    5yr   10yr
## 0.9991 1.0146 0.9962
##    2yr    5yr   10yr
## 1.0268 0.9912 1.0090
##   2yr   5yr  10yr
## 1.001 1.001 1.000
##    2yr    5yr   10yr
## 1.0048 1.0010 0.9837
##    2yr    5yr   10yr
## 1.0075 0.9881 0.9972
```

Obligatory deathstar example

```
require(deathstar, quietly = TRUE)

estimatePi <- function(seed) {
    set.seed(seed)
    numDraws <- 10000
    r <- 0.5
    x <- runif(numDraws, min = -r, max = r)
    y <- runif(numDraws, min = -r, max = r)
    inCircle <- ifelse((x^2 + y^2)^0.5 < r, 1, 0)
    sum(inCircle)/length(inCircle) * 4
}

cluster <- c("localhost")
run.time <- system.time(ans <- zmq.cluster.lapply(cluster = cluster, as.list(1:1000),
    estimatePi))

print(mean(unlist(ans)))

## [1] 3.142

print(run.time)

##    user  system elapsed
##   1.276   0.816   6.575

print(attr(ans, "execution.report"))

##              jobs.completed
## krypton:9297             84
## krypton:9300             83
## krypton:9306             83
## krypton:9308             83
## krypton:9311             83
## krypton:9314             83
## krypton:9318             84
## krypton:9325             83
## krypton:9329             84
## krypton:9332             83
## krypton:9377             84
## krypton:9380             83
```

doDeathstar foreach example

```r
require(doDeathstar, quietly = TRUE)
registerDoDeathstar("localhost")

z <- foreach(i = 1:100) %dopar% {
    set.seed(i)
    numDraws <- 10000
    r <- 0.5
    x <- runif(numDraws, min = -r, max = r)
    y <- runif(numDraws, min = -r, max = r)
    inCircle <- ifelse((x^2 + y^2)^0.5 < r, 1, 0)
    sum(inCircle)/length(inCircle) * 4
}

print(mean(unlist(z)))

## [1] 3.142
```

Thanks for listening!

Many people contributed ideas and helped debug work in progress as the rzmq package was being developed.

Bryan Lewis for collaborating and planning this talk with me.
JD Long for my excessive reuse of the estimatePi example.
Kurt Hornik for putting up with my packaging.
John Laing for finding bugs in my code.
Prof Brian Ripley for just being himself.