RESULTS VISUALISATION

RESULTS VISUALISATION

- At the beginning of this course, the large majority of respondents to the survey had no experience with visualisation/plotting tools, or found this challenging.
- Wide range of software suitable for such purposes.
- As always, ease of use is traded for the number of features supported and quality of end results.
- For CSLP you may use things as simple as charts generated with spreadsheet tools (LibreOffice Calc, Microsoft Excel, OS X Numbers, Google Sheets, etc.),
- or work with specialised software/packages/statistical computing languages (gnuplot, Matlab, matplotlib, R).

RESULTS VISUALISATION

- Like with other things, the visualisation tool is mostly a personal choice.
- Though at times you may be required to use a specific tool (employer/project request or license constraints).
- If you work collaboratively, open-source, cross-platform portable solutions are certainly appropriate.

PLOTTING WITH R

- Today I will give a short guide to **R**, since
  1. It meets the aforementioned criteria;
  2. You will likely use this tool for other projects where you will need to process and visualise data sets.
- R is increasingly popular among data analysts and statisticians.
- Workflow can be simplified with the use of graphical front-ends such as RStudio (as you type help, partial script execution, exporting images, etc.).
- As you become more expert, you can combine it with C/Java/Python code.

PLOTTING WITH R

- Once you installed R (already installed on DiCE), you can invoke different functions through a CLI.

```
$ R
...
Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```

- Though perhaps more often you will write some scripts.
- Even if using a graphical front-end, you still have the console, which is handy if you need to install packages.

R PACKAGES

- DiCE machines should have most packages you would require, but for your personal installation, you may have to install some manually.
- For instance, you may want to install the **ggplot2** package to produce complex graphics more easily.
- The procedure is pretty straightforward:

```
> install.packages("ggplot2")
```

- This takes care of all the necessary download, compilation, and installation for you.

WRITING A SIMPLE (YET USEFUL) R SCRIPT

WRITING A SIMPLE SCRIPT

- Say we have some delays you recorded in a file named 'values.dat' and you want to see if they follow a certain distribution.
- Imagine a file like this

```
delay
78
500
13
190
95
 ...
```

- Where first line contains the name of variable observed.
- We expect the delays stored in this file to follow an Erlang distribution with shape $k=2$ and rate $\lambda=1/100$.

WRITING A SIMPLE SCRIPT

- First thing we do is to read these values from the file

```
measurements <- read.delim("values.dat")
```

- Then obtain an estimate of the Probability Density Function (PDF) for the values corresponding to the 'delay' object in the dataset.

```
empiricalDistr <- density(measurements$delay)
```

- The 'density' function is implementing a kernel density estimator (though no need to worry about the details).

WRITING A SIMPLE SCRIPT

- Next we obtain an 'ideal' PDF of an Erlang-2 random variable with rate $\lambda=1/100$, where say we are interested in delays ranging between 0 and 500 seconds.

```
span <- seq.int(0, 500, length.out=500)
idealDistr <- dgamma(span, 2, rate=1/100)
```

- Here we are actually drawing from a gamma distribution, but since the shape is an integer ($k=2$) gamma and Erlang are equivalent.

WRITING A SIMPLE SCRIPT

- What remains is only to plot the two curves

```
plot(span, idealDistr, type="l",
  col="red", lwd=2,
  xlab="Delay [s]", ylab="Probability", main="PDFs")
lines(empiricalDistr, col="blue", lwd=2)
```

  where we plot with lines (of width 2), use red for the 'ideal' PDF and blue for the empirical distribution, and label the axes accordingly.

- Finally set the legend

```
legend(370, 0.0035, legend=c("Ideal", "Empirical"),
       col=c("red", "blue"), lty=1:1, lwd=2:2, cex=1.2)
```

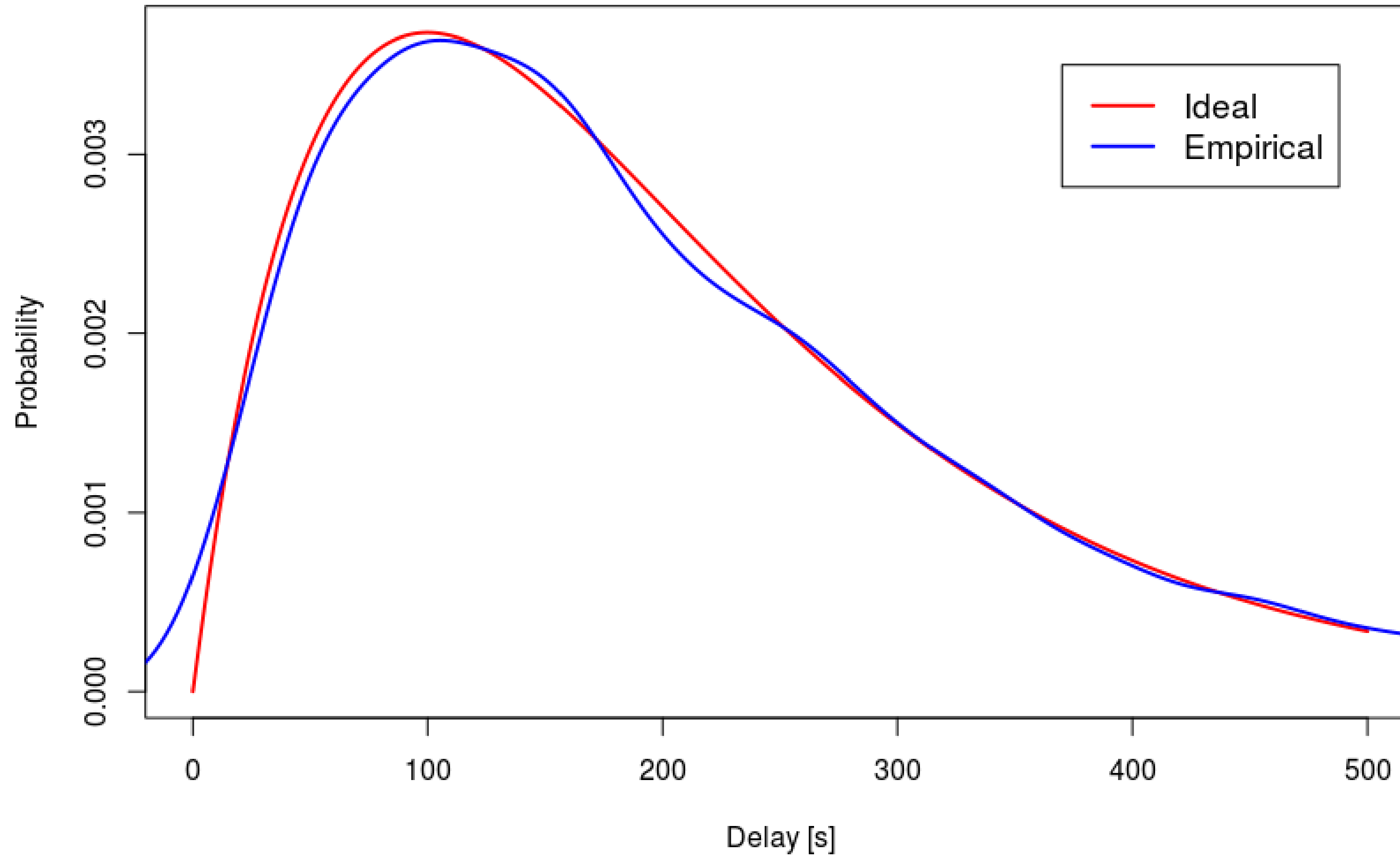
WRITING A SIMPLE SCRIPT

Putting everything together, the following script...

```
measurements <- read.delim("values.dat")

empiricalDistr <- density(measurements$delay)

span <- seq.int(0, 500, length.out=500)

idealDistr <- dgamma(span, 2, rate=1/100)

plot(span, idealDistr, type="l", col="red", lwd=2,
  xlab="Delay [s]", ylab="Probability", main="PDFs")
lines(empiricalDistr, col="blue", lwd=2)

legend(370, 0.0035, legend=c("Ideal", "Empirical"),
       col=c("red", "blue"), lty=1:1, lwd=2:2, cex=1.2)
```

WRITING A SIMPLE SCRIPT

...produces this figure



PDFs
Ideal
Empirical
Probability
0.000
0.001
0.002
0.003
0
100
200
300
400
500
Delay [s]

PRODUCING FANCIER PLOTS

- Say you want to plot the time evolution of some metric at two different agents.
- E.g. the throughput of two stations in a Wi-Fi network, when one of them changes the PHY rate.
- Data stored in a CSV file, first column time (in seconds), 2nd and 3rd column stations' throughputs (in kb/s).
- The file would look like the following:

```
10.000, 1.208e+04, 1.205e+04
11.000, 1.196e+04, 1.207e+04
12.000, 1.221e+04, 1.196e+04
13.000, 1.189e+04, 1.230e+04
14.000, 1.188e+04, 1.226e+04
15.000, 1.189e+04, 1.261e+04
...
```

PRODUCING FANCIER PLOTS

- First load the libraries needed.

```
library(ggplot2)
library(reshape2)
```

- Prepare file path and read the contents of the file.

```
folder = "results" # location of data files

# read from CSV file
# filename obtained through concatenation
contents <- read.csv(paste0(folder, "/throughput.dat"),
                header=F)
```

- Set suggestive names for the objects.

```
names(contents) <- c("time", "STA 1", "STA 2")
```

PRODUCING FANCIER PLOTS

- Create an empty data frame and combine with read data.

```
# create empty data frame
mydata <- data.frame()

# combine objects
mydata <- rbind(mydata, contents)
```

- Time logged started at 10s, so make adjustment to display more elegantly.

```
# adjust time to display
mydata$time <- mydata$time - 10
```

PRODUCING FANCIER PLOTS

Produce the plot...

```
myplot <- ggplot(melt(mydata, id="time"),
  aes(x=time, y=value/1e3)) +
  geom_line(aes(colour=variable)) +
  scale_x_continuous(limits=c(0,250)) +
  scale_y_continuous(limits=c(0,15)) +
  ylab("Throughput [Mb/s]") +
  xlab("Time [s]") +
  theme_bw() +
  theme(plot.margin = unit(c(0.5,1,0,0.5), "lines"),
        plot.background = element_blank(),
        legend.title=element_blank(),
        legend.position="top",
        text = element_text(size=20)) +
  scale_color_manual(values=c("cadetblue4", "coral4"))
```

... OK, a lot to take in here! Let's go through this step by step.

USING THE `GGPLOT` FUNCTION

- First we need to convert data object into a molten data frame, telling the plotter the variable changing is 'time'.

```
myplot <- ggplot(melt(mydata, id="time"),
...
```

- Then construct aesthetics mapping, i.e. x and y axes. We want to plot in Mb/s, so we need to divide throughput values by 1,000.

```
aes(x=time, y=value/1e3)) +
...
```

- Instruct to connect the variables in order specified by x axis, with lines; allow different colours for each.

```
geom_line(aes(colour=variable)) +
...
```

USING THE GGPLOT FUNCTION

- Set ranges for the x and y axes, and label these.

```
scale_x_continuous(limits=c(0,250)) +
  scale_y_continuous(limits=c(0,15)) +
  ylab("Throughput [Mb/s]") +
  xlab("Time [s]") +
```

- Set a simple theme, adjust the margins slightly, no background

```
theme_bw() +
  theme(plot.margin = unit(c(0.5,1,0,0.5), "lines"),
        plot.background = element_blank(),
...
```

USING THE `GGPLOT` FUNCTION

- No legend title, place the legend at the top, increase font size to improve readability

```
legend.title=element_blank(),
        legend.position="top",
        text = element_text(size=20)) +
...
```

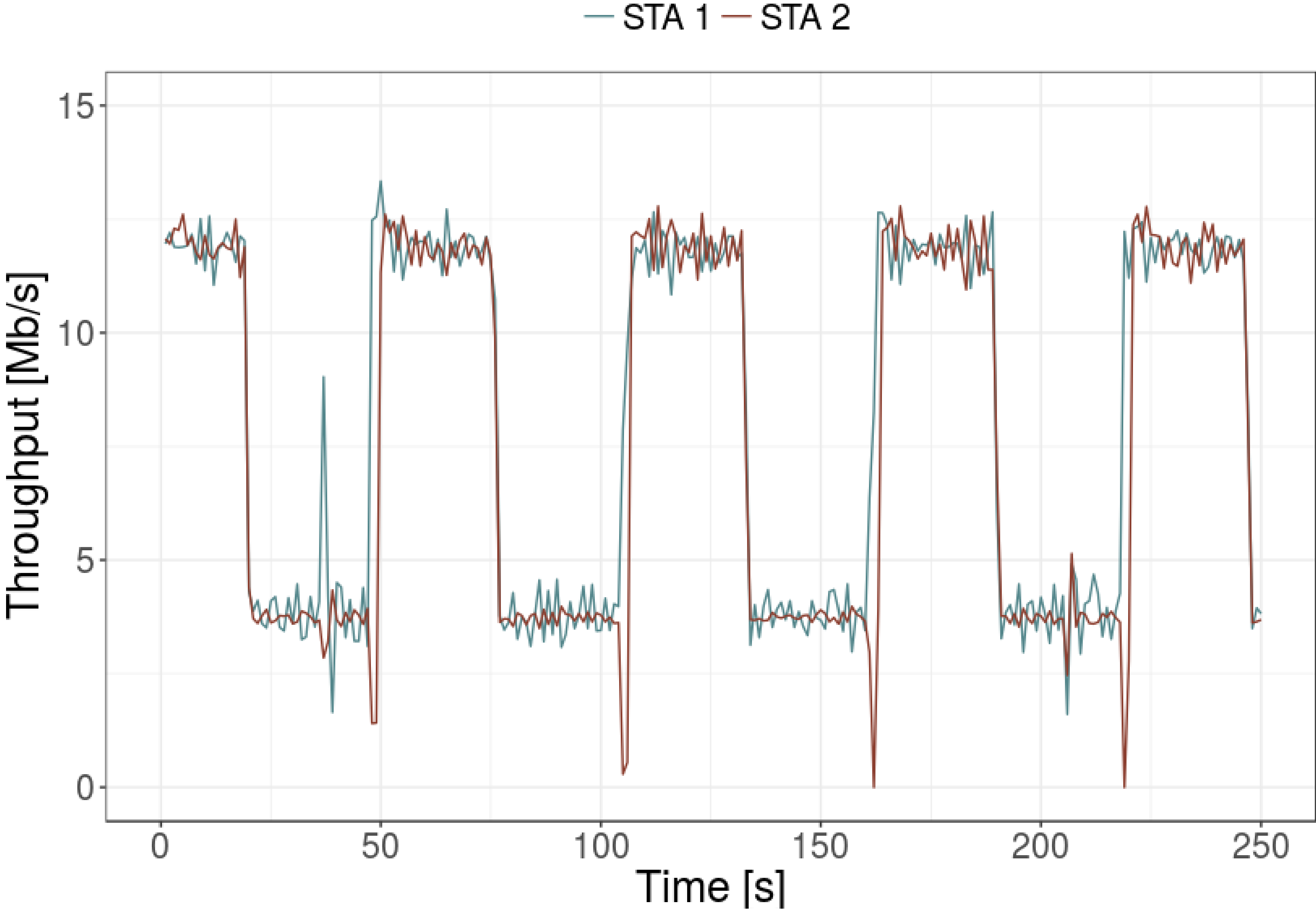
- Finally, set some **custom colours**

```
scale_color_manual(values=c("cadetblue4", "coral4"))
```

- And now plot the chart.

```
plot(myplot)
```

END RESULT



BAR PLOTS & ERROR BARS

- Now let's try something more complex.
- Say we want to compare the latency performance of two network protocols, when a client downloads files of different sizes.
- With each protocol, we download every file and measure the delay over 10 such experiments.
- We are interested in the average and standard deviation of the latency measured.

BAR PLOTS & ERROR BARS

Files containing these measurements for each protocol will look like this

```
256kB   00.23
256kB   00.19
...
512kB   00.52
512kB   00.42
...
...
4096kB  03.30
4096kB  04.29
```

BAR PLOTS & ERROR BARS

- As before, we first extract and label the data

```
mydata_1 <- read.delim(paste0(folder,
                "/latency_1.dat"), header=F)
mydata_2 <- read.delim(paste0(folder,
                "/latency_2.dat"), header=F)

names(mydata_1) <- c("FILESIZE", "LATENCY")
names(mydata_2) <- c("FILESIZE", "LATENCY")
mydata_1$what = "Protocol 1"
mydata_2$what = "Protocol 2"
```

- Then prepare empty data frames to store the average and standard deviation values of the measured latency

```
avg_lat <- data.frame()
std_lat <- data.frame()
```

BAR PLOTS & ERROR BARS

- Next, subset the data using the different file sizes

```
for(file in unique(mydata_1$FILESIZE)) {
  tmp_1 <- subset(mydata_1, FILESIZE==file)
  tmp_2 <- subset(mydata_2, FILESIZE==file)
```

- Compute the means for each protocol and bind data

```
avg_lat = rbind(avg_lat, rbind(c(what="Protocol 1",
                    FILESIZE=file,
                    LATENCY=mean(tmp_2$LATENCY)),
                c(what="Protocol 2", FILESIZE=file,
                    LATENCY=mean(tmp_1$LATENCY))))
```

- and do the same for standard deviations

```
std_lat = rbind(std_lat, rbind(c(what="Protocol 1",
                    FILESIZE=file,
                    LATENCY=sd(tmp_2$LATENCY)),
                c(what="Protocol 2", FILESIZE=file,
                    LATENCY=sd(tmp_1$LATENCY))))
}
```

BAR PLOTS & ERROR BARS

- Recall we need molden data frames for `ggplot`

```
dd_avg_lat <- melt(avg_lat, id=c("what", "FILESIZE"))
dd_std_lat <- melt(std_lat, id=c("what", "FILESIZE"))
```

- The error bars should correspond to $\mu \pm \sigma$. Therefore we need to compute the boundaries of the error bars.

```
dd_avg_lat$ymax <- as.numeric(dd_avg_lat$value) +
        as.numeric(dd_std_lat$value)
dd_avg_lat$ymin <- as.numeric(dd_avg_lat$value) -
        as.numeric(dd_std_lat$value)
```

- We should now have everything we need for plotting.

BAR PLOTS & ERROR BARS

- We want to plot the average latency with boxes (bar), as a function of the file size

```
myplot <- ggplot(dd_avg_lat, aes(x=as.character(FILESIZE),
  y=as.numeric(value))) +
  geom_bar(aes(fill=what), position = "dodge", stat="identity") +
  geom_bar(aes(fill=what), position = "dodge", stat="identity",
  colour="black") +
```

- Here we use bar chart geometry; we want to avoid overlap (hence 'dodging' bars side by side),
- Force the height of the bars to the value of the data ('identity').
- First bars will have a coloured fill, second bars black contours around these.

BAR PLOTS & ERROR BARS

- We add the error bars next, centring on the boxes, adjusting width of the ends and line.

```
geom_errorbar(aes(ymax=ymax, ymin=ymin, fill=what ),
        position=position_dodge(.9), width=0.25, lwd=1) +
```

- Custom tics on the x axis

```
scale_x_discrete(limits=c("256kB", "512kB", "1024kB",
        "2048kB", "4096kB")) +
```

- Labelling the axes

```
ylab("Download time [s]") + xlab("File size") +
```

BAR PLOTS & ERROR BARS

- Finally setting the theme, legend position, font size, etc.

```
theme_classic() +
  theme(legend.title=element_blank(),
        legend.position=c(0.15,0.9),
        legend.background = element_rect(),
        text = element_text(size=20)) +
```

- and custom colours.

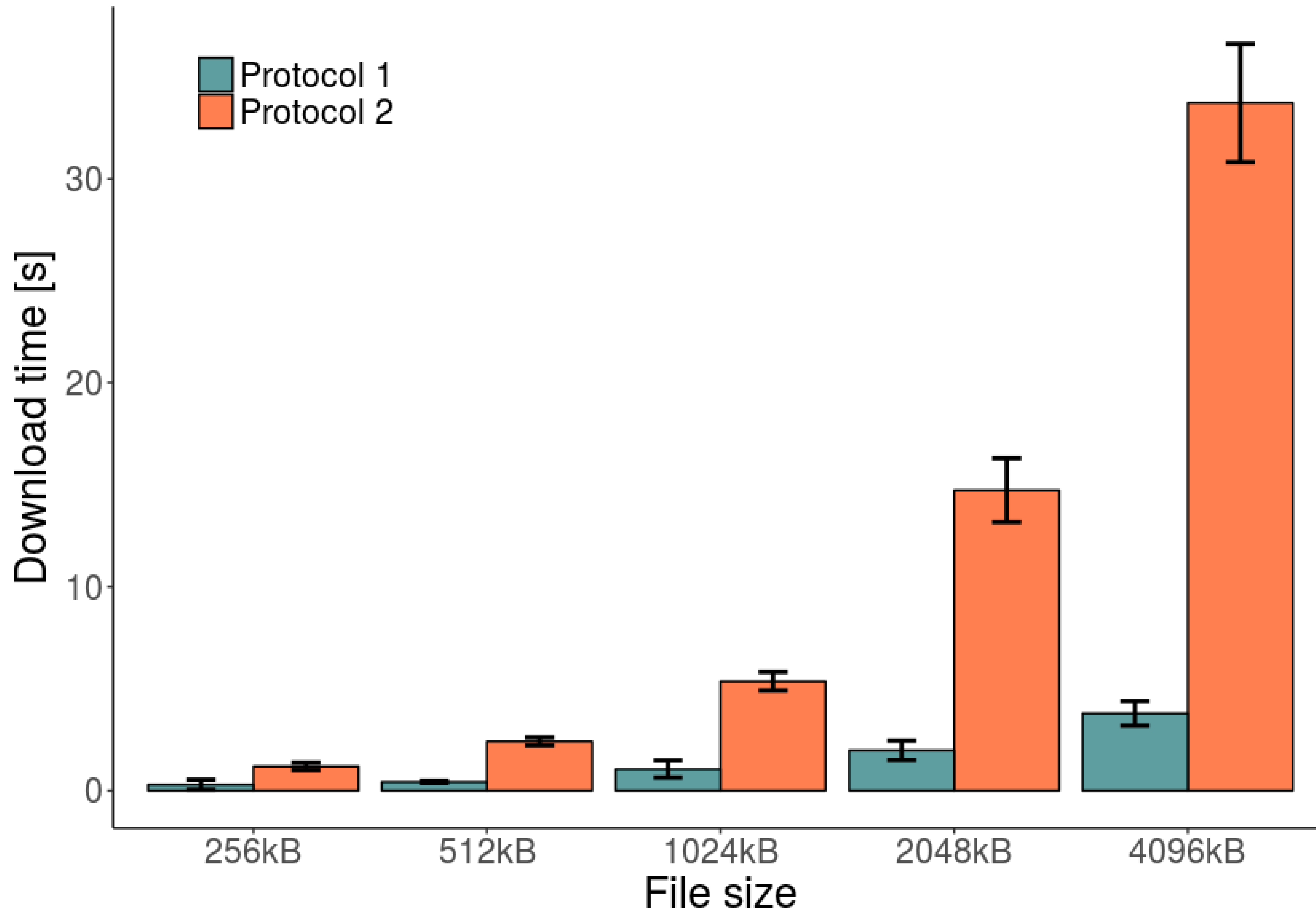
```
scale_fill_manual(values=c("cadetblue", "coral")))
```

BAR PLOTS & ERROR BARS

The complete call

```
myplot <- ggplot(dd_avg_lat, aes(x=as.character(FILESIZE),
  y=as.numeric(value))) +
  geom_bar(aes(fill=what), position = "dodge", stat="identity") +
  geom_bar(aes(fill=what), position = "dodge", stat="identity",
    colour="black") +
  geom_errorbar(aes(ymax=ymax, ymin=ymin, fill=what ),
    position=position_dodge(.9), width=0.25, lwd=1) +
  scale_x_discrete(limits=c("256kB", "512kB", "1024kB",
    "2048kB", "4096kB")) +
  ylab("Download time [s]") + xlab("File size") +
  theme_classic() +
  theme(legend.title=element_blank(), legend.position=c(0.15,0.9),
     legend.background = element_rect(),
     text = element_text(size=20)) +
  scale_fill_manual(values=c("cadetblue", "coral"))
```

…ET VOILA!



ADDITIONAL RESOURCES & REMARK

- If you feel R is for you and want to learn how to produce even more sophisticated charts, lots of tutorials and examples are available on the **R-bloggers** web site.
- Everything you need to know about `ggplot2` features is explained **here**.
- **NB:** You are not required to produce plots of the results you obtain for CSLP using R. You can use any tool you like, though R may prove useful for future projects as well.

PART 3 REQUIREMENTS

PART 3 SUBMISSISON

- Part 3 carries 50% of the total marks
- Deadline: Wed 21 st December, 2016 at 16:00
  (unless ITO/year organiser gave you an extension)
- Wise to check again and comply with the University **academic misconduct policy**.
- Remember this is an individual project.
- Reusing publicly available code is OK, but clearly mark the parts you did not author yourself.

PART 3 REQUIREMENTS

- By the part 3 deadline you are expected to have:
  - A complete and working version of the simulator.
  - A short written report (in PDF format) documenting your implementation and key findings.
- Your code will not be tested for functionality you were expected to have at Part 2, but will be expected to work on valid input files provided as command line arguments.

FUNCTIONALITY EXPECTED FOR PART 3

- Correct lorry scheduling, bin service, and route planning;
- Correct summary statistics, in line with output format specification;
- Experimentation support;
- Test files non-trivially different from each;
- Evidence of reasonable run times and optimisation;
- Appropriate indentation and spacing, no dead code;
- Evidence of appropriate use of source control.

WRITTEN REPORT

- Do not write an overly lenghty report;
- Avoid putting code; If you really need to explain something non-trivial, use an appendix;
- Briefly document the architecture of your simulator (you may already have this by now), your design choices, and testing efforts;
- Briefly discuss experiments performed, results obtained (potentially accompanied by plots), and insights gained;
- Document any additional features you implemented (extra credit for these).

QUESTIONS