Experiences with Teaching a Second Year Distributed Computing Course

Rizos Sakellariou(✉)

School of Computer Science, University of Manchester, Manchester, UK
rizos@manchester.ac.uk

Abstract. The proliferation of parallel and distributed computing in the last years has led to calls for the early introduction of parallel and distributed computing in the undergraduate curriculum arguing that the topic should and can be offered at different levels but some basic knowledge must be acquired by every computer science graduate. However, there is no widespread agreement on how this can be achieved. This paper contributes to the debate by presenting the approach and experiences from designing and teaching a second year undergraduate distributed computing course that has been running for a decade in the School of Computer Science of the University of Manchester. The course evolved to follow an approach which presents material in a way that attempts to emphasize the importance of four key pillars of abstraction, which underpin the design and management of modern distributed systems, namely: trade-offs, failures, concurrency and synchronization, performance. The paper presents the details of this approach arguing that the use of suitable abstractions allows for a rewarding learning experience that helps students familiarize with and appreciate the challenges of distributed computing at an early stage.

1 Introduction

The so-called "triumph of parallel computing" [9] is indisputable. Whether in the presence of multiple CPUs in everyday computing devices or as part of large-scale infrastructures such as Google, parallel and distributed computing manifests itself nowadays as a mainstream Computer Science topic. However, there are several indications that this trend has not fully found its way into the Computer Science curriculum, especially to the degree that would be desirable in order to train successfully future computer scientists. Motivated by such observations, a report [7], produced by a working group aiming to develop curricular guidelines for Computer Science and Engineering undergraduates, tried to identify essential topics and concepts and where in the curriculum they could be incorporated to ensure that "*all students graduating with a bachelor's degree in computer science/computer engineering receive an education that prepares them in the area of parallel and distributed computing, preparation which is increasingly important in the light of emerging technology*" [7]. Despite the elaborate discussion of how and where different concepts need to be introduced, the report did not really

F. Desprez et al. (Eds.): Euro-Par 2016 Workshops, LNCS 10104, pp. 28–37, 2017.
DOI: 10.1007/978-3-319-58943-5_3

make concrete suggestions for specific new courses tailored towards an early introduction of parallel and distributed computing in the curriculum.

Traditionally, parallel and distributed computing courses have been considered as advanced courses taught at advanced undergraduate or postgraduate level. To a large extent, this reflects the historical development of parallel and distributed computing and the long track record of specialized research in the area[1]. As a result of the need to prepare graduates better in terms of parallel and distributed computing background, one may be tempted, as in [7], to identify the key concepts that need to be introduced early in the curriculum. However, as some of these concepts have been developed as part of previous specialized research, especially at a time when parallel and distributed computing was not a mainstream topic, the question is whether they carry the stamp of their time and origins, which makes them not easily amenable to early undergraduates. In other words, the question is whether educators need to think about some extra level of abstraction to introduce early undergraduates to parallel and distributed computing without risking losing them through early exposure to low-level 'heavyweight' details. Such a level of abstraction can also address the requirement to teach students principles that "*will remain relevant and abiding for the unforeseeable technologies of the next decade*", a need that was eloquently identified in [3].

As part of a curriculum redesign more than ten years ago, it has been decided to introduce Distributed Computing to the second year programme of the primarily 3-year degree of the School of Computer Science of the University of Manchester, starting Spring 2007. The author (along with a team of two colleagues) has been responsible for the initial design of the course and since then he has delivered it (jointly or as the sole instructor since 2012) every spring with the exception of 2009 when he was on sabbatical leave. In total, over 1100 students have taken the course during these ten years. Following this experience, this paper will present:

- a detailed view on how the course has been organized and delivered;
- the approach followed/developed, which tried to emphasize four main pillars at a high level of abstraction, namely: trade-offs, failures, concurrency and synchronization, performance; and,
- some observations and suggestions stemming from the 10-year experience.

The rest of the paper is structured as follows. Section 2 presents the syllabus including lecture topics and laboratory assignments. Section 3 presents the main abstraction pillars used to introduce the course. Section 4 discusses observations from the 10-year experience. Finally, Sect. 5 concludes the paper.

[1] Parallel and Distributed Computing is treated here as a single field, same as in [7] and in line with most modern views and classifications. However, it should not be forgotten that parallel (primarily aiming towards high-performance) computing on one hand and distributed computing on the other have been kept apart for many years, each with its own history and research; it is only in recent years that common aspects and principles have (re)gained importance leading to some sort of convergence.

2 Syllabus

2.1 Background

The course is offered as part of the 2nd year program of the School of Computer Science of the University of Manchester during the Spring semester, which runs for twelve weeks (interrupted by the Easter break), typically from late January to the beginning of May. The method of study consists of twenty-two 50-minute lecture slots and five 2-hour laboratory sessions, where students work on specific assignments. Assessment is based on a 2-hour closed book examination, which counts for 80% of the overall mark, and marking of the laboratory assignments counting for the remaining 20%. In recent years about 150 students register for the course each year, reflecting the rather large size of the Computer Science intake of the University of Manchester. The majority of these students study towards a 3-year BSc degree in Computer Science.

2.2 Lecture Topics

Fourteen of the lecture slots are used to introduce and discuss the course topics. These topics broadly cover issues related to: (i) motivation for parallel and distributed systems; (ii) communication; (iii) naming; (iv) synchronization (including logical timing and coordination); (v) failure management; (vi) advanced topics. The remaining slots are used for guest lectures, introducing and motivating laboratory work and revision. Lectures are supported by handouts and references to two core textbooks [4,10].

Most specifically, the fourteen slots are dedicated to the following topics:

1. *Introduction to Distributed Computing:* In addition to providing the rubric for the course, the lecture is used to motivate the challenges of distributed systems using for this the so-called "eight fallacies of distributed systems" [2]. Although more than twenty years old by now, these fallacies have an important educational value as they provide a wealth of opportunities to introduce and motivate the challenges of developing distributed systems.
2. *Introduction to Parallel Computing:* The lecture is used to introduce the need for performance and the concept of scalability. Application examples are presented and Amdahl's law (despite criticism [9]) is used as a way to highlight the impact of inherently sequential parts of an application. At the same time, alternative approaches to get an upper bound on parallelism using the work of the critical path are discussed and illustrated with scientific workflow applications that appear to gain momentum as an important class of applications that benefit from parallelism and large-scale distributed computing [11].
3. *Models and Architectures:* The focus of the lecture is on different architectures and models to build distributed systems. The trade-offs of different solutions, such as client-server versus P2P, are discussed and used to highlight the observation that there is not a single architectural solution that fits everything.

4. *RPC and RMI:* The lecture introduces Remote Procedure Calls (RPCs) as a way to provide a high-level alternative to low-level send and receive constructs. The context in which the first practical implementations of RPC were developed as well as the criticism often made are mentioned and used to understand how current solutions and problems may inherit the legacy of the past. In addition, as the Computer Science syllabus in Manchester is using Java as the introductory programming language in the first year, RMI is described as the Java approach to RPC.
5. *Exercises on RPC and RMI:* These exercises attempt to consolidate understanding but also to introduce more specialized topics such as 'call by copy/restore', at-least-once and at-most-once semantics and contrast between RPC and messaging.
6. *Name and Directory Servers:* This lecture discusses naming issues following the Internet Domain Name System as a running example.
7. *Time and Logical Clocks:* This lecture discusses approaches to deal with the lack of global clock in distributed systems. Lamport clocks and Vector clocks are introduced and their properties are explained and discussed.
8. *Coordination and Agreement:* This lecture is used to motivate and describe two classical algorithms for election of a coordinator such as ring-based election and the Bully algorithm.
9. *Transactions:* This lecture is used to introduce the problems that arise from unexpected failures and the importance of treating some operations as atomic operations using as a motivating example the problems that may occur during a bank transfer if a crash occurs half way through the transfer. The four key properties of transactions, known with the acronym ACID, are introduced, and there is a discussion of how transactions can be implemented.
10. *Distributed Transactions:* This lecture describes in more detail the issues with respect to the implementation of transactions and highlights the need to strike a balance between enforcement of transaction constraints and concurrency. Furthermore, the lecture motivates distributed transactions, describes protocols such as one-phase and two-phase commit and discusses problems that may arise in this context such as distributed deadlocks and how they can be detected.
11. *Byzantine Fault Tolerance:* This lecture is used to introduce students to some key results on arbitrary (byzantine) failures. The impossibility of making an agreement when there are three generals (one of whom is a traitor) as well as a protocol for an agreement when there are four generals, three of whom are loyal, are illustrated using examples from the original 1982 paper [5]. Occasionally, the issues with Byzantine failures are illustrated in the class by assigning the role of loyal or traitor general to groups of students and let them emulate the process of trying to make an agreement.
12. *Replication:* This lecture discusses the benefits as well as the cost of replication and describes some fundamental consistency models, such as sequential consistency, causal consistency and eventual consistency. Algorithms for placing replicas are also discussed.

13. *The Integration Game:* This lecture focuses on interoperability and standardization as mechanisms that enable the development of large systems primarily focusing on a service-based model.
14. *Cloud Computing and advanced topics:* This lecture traces back the historical development of Grid computing and the move to the Cloud. It highlights problems stemming from the sheer size of data that a number of applications produce as well as the distributed collaborative nature of such applications.

Two guest lectures are typically offered every year (subject to availability). One of the guest lecturers typically comes from the banking sector and helps students understand how some of the concepts discussed in the course are present in the problems faced by a demanding, large-scale, real-world environment. Another guest lecturer with expert knowledge has often been invited to discuss more advanced topics: in early years in relation to grid computing and e-Science, in recent years in relation to Cloud Computing and virtualization.

2.3 Laboratory Assignments

There are three laboratory assignments, each one serving a different purpose and addressing different aspects of the course, exposing students to different concepts. The assignments have been repeated every year with only minor changes or adjustments.

The first assignment is a simple exercise whose only purpose is to introduce students to three different mechanisms for implementing a client-server interaction. The three mechanisms are sockets, Java RMI and Java servlets. The choice has partly been driven by the particular study programme which uses Java as an introductory programming language. This is a simple exercise but it helps introduce concepts that are not taught at an earlier stage.

The second assignment is the main laboratory exercise, counting for 60% of the laboratory mark. In this exercise students are asked to write a client to interact with a server through HTTP and XML and make a booking. Each student's client competes with other students' clients for the booking. The idea is that a server advertises 200 slots for a band and 200 slots for a hotel and a client is asked to find the earliest common slot to arrange both a band and a hotel for a wedding as soon as possible. The server introduces random failures and delays, which requires students to build robust clients capable of dealing with such problems if they arise. Bookings are made and changed all the time, which suggests that any information on availability of slots may be quickly out-of-date and a slot that appears to be free may already be taken before a request to book it is issued. To avoid having a number of clients monopolize the server, students are asked to include a deliberate delay of one second between successive requests and they are limited to two bookings for each of band and hotel. The use of a controlled environment for the server facilitates the introduction of delays and random failures (e.g., server unavailable) that help students realize the importance of building robust code.

The third assignment asks students to build a simple discrete-event simulator that simulates a queue-based system where: (i) requests arrive at a given rate; (ii) each request requires a certain amount of server time; (iii) a number of servers can serve requests from a queue that initially stores requests; and, (iv) the size of the queue is limited. For given values of each of these four parameters, students are asked to calculate over a period of time: (i) average queue size; (ii) average response time for requests; and, (iii) percentage of requests rejected. This exercise helps students appreciate the value of performance modelling and simulation as a mechanism for capacity planning when building a distributed system.

3 Objectives and Key Abstraction Pillars

The main intended learning outcome of the course, as advertised to students, is to make them aware of the principles, techniques and methods involved when dealing with distributed systems. There are four key abstractions along with a number of lesser ones that summarize the key principles emphasized throughout the course. Following the experience over the years it has emerged that an emphasis on these abstractions would create the right mindset to help students from an early level to acquire a good understanding of the issues in distributed systems.

Trade-Offs in the Design of Distributed Systems: An important message for an early course on parallel and distributed computing is to infuse the concept of trade-off between different objectives that need to be considered when designing, building and using distributed systems. Especially for distributed computing in the early years of the curriculum it is important to make clear that such trade-offs exist and emphasize their main direct implication: there is not a single solution that fits everything. Although some of the trade-offs may be relatively obvious or, in their general form, they may have been around for long enough (e.g., the interplay between cost and performance), it appears that less obvious trade-offs emerge. Taking into account the increasing heterogeneity of modern platforms, the different options made available to users and the scales envisaged for tackling new problems, one could argue that even more interesting trade-offs will emerge in parallel and distributed computing. One needs only consider frequency scaling, a technique commonly used nowadays, and ponder about how it affects the interplay between energy, cost, and performance, in not easy to predict ways [6,8]. This suggests that creating a mindset directed towards the detection and appreciation of such trade-offs is particularly important as an essential parallel and distributed computing skill for future computer scientists. The first example of a trade-off that is used early enough in the module to motivate students is a consequence of one of the eight fallacies, "latency is not zero", and relates to the interplay between latency and bandwidth. It can be easily realized that over a low-latency network the time to send several remote requests will primarily be affected by bandwidth; in this case, many small-sized requests will result

in a good execution time. However, over a high-latency network, latency will dominate and many remote requests may result in long execution times because of the high latency; in this case, it can be faster to send one large-sized request to avoid the cost of high-latency. Interestingly enough, such an issue received high-profile attention as recently as 2006 in the context of a popular browser, leading to the suggestion that "computing over a high-latency network means you have to bulk up" [1].

Dealing with Failures: There is an old quote by Leslie Lamport that suggests that "a distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable". The message that effective distributed systems need to address failures and the primary approach to do this is some sort of redundancy is repeatedly made during the course. The second laboratory assignment helps reinforce this message by getting students realize that client code that may work once is not necessarily robust enough to deal with everything that may go wrong. It is often the case that several students realize the importance of managing unpredictable situations during this laboratory exercise, something that suggests that appreciating the importance to deal with failures requires exposure to failure handling both from a theoretical and from a practical point of view.

Implications of Concurrency: Concurrency is introduced very early in the course, primarily focusing on embarrassingly parallel applications as they provide an easy way to demonstrate the benefits of parallelism and some of them are easily recognizable by students (e.g., online games). The implications of concurrent execution are discussed as a means to motivate transactions following a classical example of concurrent accesses to the same banking accounts. This allows for a good discussion of the trade-off between synchronization and the amount of parallelism available, something which is also discussed in the context of the second laboratory exercise where excessive locking would help with individual implementations but at the same time it would reduce the overall degree of parallelism in the system. It is argued that such a simple example-driven approach helps illustrate parallelism/concurrency step-by-step without overloading students with the complexity of specialized application examples often seen in the delivery of traditional high-performance computing courses. If anything, the argument is that, in the early curriculum, the illustration of concurrency and its implications or trade-offs (especially the trade-offs and interplay between concurrency and computation versus communication or synchronization) requires even more simple examples that what are currently offered by standard textbooks.

The Quest for Performance: Performance has been a key objective in the traditional parallel computing curriculum, however, it has been often downgraded in distributed computing. Nevertheless, performance objectives are always present

in reality[2] and they are emphasized in different parts of the course, also in terms of capacity planning as part of the third laboratory assignment.

4 Discussion

The approach that was described above in organizing the course seems to work well. The overall student assessment of the course has been positive, consistently among the best of the year. Performance during the final examination, based on a mix of bookwork and small problem solving exercises, seems to indicate that a number of students get a good understanding of some key concepts related to distributed computing. Judging from the experience over the last decade, it appears that, in recent years, students come with an already improved understanding of distributed systems, which could be partly a result of the everyday use of many widespread distributed applications or smartphones. As a matter of fact, in early years the average mark (grade) of the students attending the course was significantly lower than their overall average mark for the year. In recent years, marks have improved (without any significant changes in the style or difficulty of the examination paper) and the average mark of the students attending the course has been broadly in line with their overall average mark for the year. This observation raises the bar with respect to the new topics and angles that have to be adopted to keep the course interesting and relevant on one hand but also to prepare students for future challenges in relation to parallel and distributed computing.

The generation of an ever increasing amount of data and the suggestion that many future distributed applications may need to handle large volumes of data raises a question on what is the best approach to introduce students to the challenges involved. During the course, there are only passing references to the volumes of data that may have to be handled in the future. This may have to followed-up by more advanced courses, especially as anecdotal evidence seems to suggest that addressing large-scale software requirements seems to be a skill much desired by industry. An additional issue that is also mentioned in passing is the issue of economy. The ever increasing availability of resources (memory, storage, etc.) often tempts programmers to write code without consideration of the amount of resources consumed. Even though earlier generations were used to face limited resources this skill seems to have gone away with the dramatic increase in the amount of resources. Yet, the increasing importance on issues such as energy minimization may suggest that resource-efficient programming will become more important in the future than it is now. This may have to be introduced in an early parallel and distributed computing course as an additional dimension whose trade-offs need to be considered.

[2] A guest lecturer from industry once mentioned how important and time-consuming it had been to find quick solutions in his company to ship large files between the UK and USA. Eventually the answer relied on tailored versions of TCP/IP but the solution was found through third-party products after a significant amount of time had been spent internally.

During the course, an attempt is made to give an historical perspective wherever possible. This is useful to understand the limitations of some approaches, which may have been excellent solutions at the time they were devised but may be holding understanding back in today's settings. Some examples were already given. It can be argued that giving a good historical perspective and associating different algorithms and techniques with problems and context at the time they were developed enables students to see beyond earlier limitations. In one particular case, however, when discussing consistency models, it appears that the common notation used to illustrate examples of different protocols (also adopted by [4,10]) carries lots of weight from distributed shared memory days and, every year it becomes one of the most puzzling topics for students. The question is to what extent a different notation that presents examples at a coarse-grain level (as opposed to the standard fine-grain, variable level currently used) could make things easier to understand.

An issue that clearly merits some discussion in a distributed computing course is security although one would expect that security issues are primarily addressed by specialized courses. In early versions of the course there was one lecture on security, which was dropped due to time limitations and because it was not necessarily good value for the time spent. However, one pertinent issue that has been discussed as part of the second laboratory exercise was denial of service attacks, as many students inadvertently did not add the requested one second delay in their code between successive requests to the server; such cases were monitored and, when detected, were used to discuss the problem of denial of service attack as well as the importance of capacity planning, that is planning server resources in line with the number of expected requests over a period of time.

Finally, it is interesting to trace how the course has evolved over the years to reflect experience gained, and student background in addition to changes in the field (most notably the transition from grid computing to cloud computing). The early design of the course tried to come up with a set of mostly self-contained lectures that described selected algorithms from the mainstream distributed computing literature (as an initial reference point the fourth edition of [4,10] were used). Over the years, a connecting link was developed, which crystallized in the key abstraction pillars described in Sect. 3, but most of the initial lectures remained largely unchanged. However, students in recent years started raising more often questions that added extra dimensions of complexity to some of the classical algorithms (e.g., byzantine problems with communication uncertainty, ring-based election in the presence of failures and so on). At the same time, it is the author's impression that many concepts or algorithms (such as logical clocks or election algorithms) were more easily understood and digested by students in recent years than they were in early years. This observation raises the question of whether there is scope for educators to enhance the description of some of the problems addressed by the relevant algorithms (some of which, even though classical, are a few decades old) to improve the student learning experience and possibly match some of today's problems more closely.

5 Conclusion

This paper has presented the structure of a second year distributed computing course offered by the School of Computer Science of the University of Manchester for the last decade. The approach tried to build on four main pillars at a high-level of abstraction, which allowed the introduction of a number of distributed computing topics at an early level in the curriculum. This suggests that in order to appreciate the growing challenges of the field and prepare future computer scientists well, early introduction of parallel and distributed computing in the curriculum may benefit from the introduction of and emphasis on suitable abstractions as part of appropriately (and holistically) designed courses more than simply moving some advanced (and possibly difficult to understand without suitable context) concepts to different parts of the early curriculum.

Acknowledgements. The author would like to thank Chris Kirkham and Dean Kuo who were members of the team that designed the original course. Chris, in particular, who jointly delivered the course until 2011 when he retired, and his long experience in computing contributed enormously in the early years of the course. Special thanks are also due to a number of Teaching Assistants who assisted with laboratories, the guest lecturers, and, above all, the over 1100 students who attended the course during the last ten years and helped shape it and evolve. Finally, detailed course information is available from: http://studentnet.cs.manchester.ac.uk/ugt/COMP28112/.

References

1. https://blogs.msdn.microsoft.com/oldnewthing/20060407-25/?p=31613/
2. https://en.wikipedia.org/wiki/Fallacies_of_distributed_computing
3. Adams, J., Brown, R., Shoop, E., Patterns, E.: Compelling strategies for teaching parallel and distributed computing to CS undergraduates. In: 27th IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), EduPar 2013 (2013)
4. Coulouris, G., Dollimore, J., Kindberg, T., Blair, G., Systems, D.: Concepts and Design, 5th edn. Addison-Wesley, Boston (2011)
5. Lamport, L., Shostak, R., Pease, M.: The Byzantine generals problem. ACM Trans. Program. Lang. Syst. (TOPLAS) **4**(3), 382–401 (1982)
6. Pietri, I., Sakellariou, R.: Cost-efficient CPU provisioning for scientific workflows on clouds. In: Altmann, J., Silaghi, G.C., Rana, O.F. (eds.) GECON 2015. LNCS, vol. 9512, pp. 49–64. Springer, Cham (2016). doi:10.1007/978-3-319-43177-2_4
7. Prasad, S.K., Gupta, A., Kant, K., Lumsdaine, A., Padua, D., Robert, Y., Rosenberg, A., Sussman, A., Weems, C.: Literacy for all in parallel and distributed computing: guidelines for an undergraduate core curriculum. CSI J. Comput. **1**(2), 81–95 (2012)
8. Rauber, T., Rünger, G.: Modeling and analyzing the energy consumption of fork-join-based task parallel programs. Concurr. Comput.: Pract. Exp. **27**(1), 211–236 (2015)
9. Schreiber, R.: A few bad ideas on the way to the triumph of parallel computing. J. Parallel Distrib. Comput. **74**(7), 2544–2547 (2014)
10. Tanenbaum, A.S., Van Steen, M.: Distributed Systems: Principles and Paradigms, 2nd edn. Prentice-Hall, Upper Saddle River (2006)
11. Taylor, I.J., Deelman, E., Gannon, D.B., Shields, M.: Workflows for e-Science: Scientific Workflows for Grids. Springer, Heidelberg (2014)