Preface

This book presents a practical and rigorous method to write correct distributed programs. A *distributed* program, also called a *concurrent* or *multi-threaded* program, is executed simultaneously by multiple interacting threads that may be spread over more than one computer. An example is a distributed web program in which a thread executing a browser program interacts with a thread executing a server program. In contrast, a *single-threaded* program is executed by a single thread; it interacts with the environment only at the start and at the end.

Distributed programs, originally present only in operating systems, are now everywhere. Practically every interactive digital system, from a digital camera to a computer network, is a realization of a distributed program. Most programming languages now support distributed programming, providing constructs to create threads to execute functions and constructs to synchronize threads (e.g., semaphores, locks, and condition variables).

Writing *correct* distributed programs is not easy because, in addition to the difficulties of single-threaded programming, one has to account for all possible variations in the execution speeds of threads (i.e., race conditions). This book uses the well-established discipline of *assertional reasoning* for this purpose. It is worth noting that for a single-threaded program, assertional reasoning reduces to annotating the program with pre- and postconditions and loop invariants, a discipline usually covered in introductory programming courses.

A related difficulty is in achieving *compositionality*. A program is usually a collection of interacting component programs. Consider a program B that interacts with a program A. In order to write B without delving into the internals of A (regardless of whether or not A's source code is available), one needs a definition of the intended external behavior of A. We refer to this as the *service* of A. The service of A plays two roles: (1) it is used instead of A when writing B and (2) it defines the conditions that A, or any program that *implements* the service, must satisfy. If A is single-threaded, its service simply defines A's outputs in terms of its inputs. But if A is multi-threaded, it is not so simple. The service must define all possible outputs of A for every possible sequence of past inputs and outputs, and do this in a way that is usable for the two roles mentioned above.

In this book, services are defined by *service programs*, which are programs with special structure and powerful synchronization constructs. These features make a service program easy to understand. Because it is a regular program, it can be directly used as a component when writing other programs. We show how to prove that a program implements a service.

Most books on distributed programming focus on a particular distributed programming language. They describe how to write distributed programs in that language, but not how to ensure that the programs are correct regardless of variations in thread speeds. They also do not give a way to define services. In particular, they characterize the external behavior of a program *A* by a so-called interface, which states the signatures of *A*'s public functions. But this is not adequate as a service: the signature of a function does not relate its outputs to inputs, nor does it define all the situations when the function can be called.

There are books that rigorously address both difficulties mentioned above. They have methods to account for all possible variations in thread speeds, and they have methods to define services. However, they typically use a programming notation that hides threads. A program is written as a collection of atomic actions any of which can be executed at any time (perhaps constrained by a boolean guard). Thread creation and thread synchronization are not explicit. As a result, their programs and proofs are very elegant, but translating the programs into a real programming language is not straightforward. Perhaps this is why these methods are not commonly used by programmers.

The purpose of this book is to explain a practical method to write services of distributed programs and to write correct distributed programs that can make use of services. The method, called SESF (for "Systems and Services Framework"), uses a realistic programming notation, so that whatever is practiced in the book can also be done easily with real-life distributed programming languages (such as C, C++, Java, Python).

This book presents numerous applications of SESF to problems in distributed computing and networking. It consists of three parts. The first part introduces SESF by examples of gradually increasing complexity: simple locks, bounded buffers, channels, and sliding window protocols. The second part presents the theory underlying SESF. The third part consists entirely of applications to classical problems in distributed computing, including locks, termination detection, object transfer, shared memory, network sockets, and transport protocols. There are exercises throughout, some straightforward and some quite sophisticated. Solutions are available on the book's website.

An undergraduate course can easily cover the first part, preferably accompanied by coding the examples in a real language, say Python. A graduate course can cover most, if not all, of the book. Professional programmers can use the book to learn an effective way to think about distributed programs.

Acknowledgments

This book has suffered a long and torturous gestation, involving at least one reincarnation. As such, there are many people who have influenced the book, directly or indirectly. There are some I would like to thank specifically.

Simon Lam and Jayadev Misra got me started in this area long ago and supported me in my efforts over the years. Simon was my Ph.D. advisor; he introduced me to computer networking protocols and their compositional verification. Jayadev introduced me to distributed programming and assertional reasoning. They have had a profound, albeit indirect, influence on this book.

Tamer El-Sharnouby, a former Ph.D. student, showed me how to bring SESF into the context of real programming languages. He wrote service programs in Java for several network protocols. He developed a distributed harness for testing a protocol implementation against its service program, and used it in an undergraduate computer networks course. His work convinced me to make threads explicit, which resulted in a new incarnation of SESF.

Kirsten Stephen and Jeff Stuckman provided valuable feedback when they were students in a graduate class in which I used an earlier draft of this book. Kirsten combed the material like a crime scene, exposing loose ends and dubious accounts. Jeff taught me a devious property about distributed timestamps that I had never suspected.

Bobby Bhattacharjee has provided keen insights and advice at various points during the writing of this book. He also suggested the title of the book, a drastic improvement over the title I originally had in mind. Hanan Samet has been a source of inspiration and support throughout my years at UMD, especially in difficult times. Larry Davis has kindly spared me from certain administrative and teaching tasks during the book's interminable "almost finished" state. Ashok Agrawala has generously made me a co-conspirator in many of his endeavors over the years, expanding my horizons in unsuspecting ways.

Carol Whitney, my wife, has given me ample emotional and intellectual support. Early on, she read *several* drafts, giving crucial advice on what worked and what did not. Near the end, she exhorted me to put the book out of its misery before I get further behind. Throughout, she has taken me to beautiful places with elegant birds.

Maryland, USA A. Udaya Shankar

http://www.springer.com/978-1-4614-4881-5

Distributed Programming
Theory and Practice
Shankar, A.U.
2013, XVIII, 386 p.,
ISBN: 978-1-4614-4881-5