CAMBRIDGE

Cambridge University Press
0521607558 - Protocols by Invariants
Anneke A. Schoone
Excerpt
More information

1

Introduction

In the past two decades, distributed computing has evolved rapidly from a virtually non-existent to an important area in computer science research. As hardware costs declined, single mainframe computers with a few simple terminals were replaced by all kinds of general and special purpose computers and workstations, as the latter became more cost effective. At many sites it became necessary to interconnect all these computers to make communication and file exchanges possible, thus creating a computer network. Given a set of computers that can communicate, it is also desirable that they can cooperate in some sense, for example, to contribute to one and the same computation. Thus a network of computers is turned into a distributed system, capable of performing distributed computations. The field of distributed computing is concerned with the problems that arise in the cooperation and coordination between computers in performing distributed tasks.

Distributed algorithms (or: protocols) range from algorithms for communication to algorithms for distributed computations. These algorithms in a distributed system appear to be conceptually far more complex than in a single processing unit environment. With a single processing unit only one action can occur at a time, while in a distributed system the number of possibilities of what can happen when and where at a time tends to be enormous, and our human minds are just not able to keep track of all of them.

This leads to the problem of determining whether the executions of a distributed algorithm indeed have the desired effect in all possible circumstances and combinations of events. Testing algorithms has now become completely infeasible: some form of "verification" is the only way out. But verifying a distributed algorithm by verifying all possible executions still is infeasible. Hence the need for a more formal approach that aims at assessing properties of an algorithm which hold for

© Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
0521607558 - Protocols by Invariants
Anneke A. Schoone
Excerpt
More information

all possible executions simultaneously. This is the course taken in *assertional verification by system-wide invariants.*

Although a lot of work has been done in devising proof systems for distributed algorithms with desirable properties such as modularity and compositionality, our approach is entirely pragmatic.

1.1 Distributed Computing

A distributed computation is not so much a computation in a distributed system, as a computation which is distributed in a system. We will first discuss what distributed systems look like in section 1.1.1, and then give the model and notation that we use to describe a distributed computation in such a system (section 1.1.2).

1.1.1 Distributed systems

To deal with the complexity of distributed systems, computer networks are designed and implemented as layered structures. (See for example the books by Tanenbaum [Tan88], Halsall [Hal85], and Sloman and Kramer [SK87] on this subject.) In the hierarchy of layers, each *layer* implements a set of functions in order to provide a service to the layer above, making use of the services provided by the lower layers, without any knowledge of how these services are implemented. The set of rules governing the interaction between two layers in one network node is called an *interface*, while the set of rules concerned with the interaction within one layer between network nodes is called a *protocol.* Thus most distributed algorithms can be called protocols.

The layered structure simplifies the implementation and maintenance of a distributed system, due to the separation of concerns between layers. The system is also more flexible, since one layer can be changed by itself, provided the interfaces remain the same. The layered structure has also led to standardization: since 1977 the International Standards Organization (ISO) has been developing a full reference model for the communication system, the main purpose being to make communication between computers from different manufacturers possible. This model consists of seven layers, and is briefly discussed in section 1.1.1.1.

In section 1.1.1.2 we discuss how we model a distributed system, and in section 1.1.1.3 we give an overview of the assumptions about the network that we encounter in the design of distributed algorithms.

1.1.1.1 The ISO reference model. The ISO reference model is also called the Reference Model for Open Systems Interconnection (OSI). The term "open" means

© Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
0521607558 - Protocols by Invariants
Anneke A. Schoone
Excerpt
More information

that systems which follow the standards in this model are open to other systems that adhere to the same standards. The reference model has been set up to provide standards for communication between mainframes of different manufacturers, and is currently extended for arbitrary distributed systems. The seven layers identified in the model are (starting below in the hierarchy): the physical layer, the data link layer, the network layer, the transport layer, the session layer, the presentation layer, and the application layer. In each layer, the entities sent can be different, both in size and in added control information, such as address information, error-detecting codes, and sequence numbers. If it is not necessary to make a distinction between these entities, we will just call them "messages".

The *physical layer* is concerned with the transmission of *bits* over a physical circuit. Issues in this layer are, for example, by what electrical signal a bit is represented, and what kind of plugs and sockets are used.

The *data link layer* turns the raw physical circuit into a link that appears as an error-free connection to the network layer. The entities transmitted in this layer are usually *frames*, as opposed to single bits. It performs error and flow control: frames can get lost or garbled and, as it may be necessary to retransmit frames to remedy this, one also has to deal with frames that arrive out of order or that are duplicates of frames that have arrived already.

The *network layer* deals with switching and routing issues. The units that are transmitted are usually *packets*. It is the responsibility of the network layer to deliver every packet sent at its destination, wherever that may be. Thus the question of how to determine a route for every destination from a given source is an important issue in this layer. In this layer it can be assumed that packets do not get lost, garbled, duplicated, or reordered on a single link, as the data link layer is supposed to filter those errors out.

The *transport layer* is concerned with setting up and closing *connections* between two nodes in the network, and doing this in the most efficient way. This is the first *end-to-end* layer: communication is between arbitrary network nodes, whereas in the lower layers there is only communication between neighbors. In this layer, it is again necessary to consider loss, duplication, and reordering of messages. This is due to the fact that, although the network layer is supposed to provide a new route in case a link on the way goes down, it cannot prevent message loss due to the link failure, or duplication if a supposedly lost message is retransmitted over the new route, or reordering of messages if the new route proves to be faster than the old route.

The *session layer* is to provide *sessions* between application entities, for example a transaction between a banking terminal and the bank's computer. Authentication might be required here. The session layer is often merged with the transport layer.

© Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
0521607558 - Protocols by Invariants
Anneke A. Schoone
Excerpt
More information

The *presentation layer* deals with the representation of information, and the necessary formatting and conversions. Examples are data compression, data conversion, e.g. between different character codes, and data encryption, to provide security. It also deals with the fact that terminals can differ widely in for example line and screen length used, and the use of special characters.

The *application layer* is the highest layer, and deals with the communication between software systems and user programs at different network nodes. Examples are distributed databases and electronic mail systems.

1.1.1.2 Modeling a distributed system. As yet there is no universally accepted definition of a distributed system. After the informal description above, we will now give a more abstract definition, related to our model of distributed computations.

> A *distributed system* is a set of processes which communicate by message passing.

It is also possible that processes communicate by shared memory, but we do not consider this here. However, we model the "communication by message passing" by shared variables, albeit in a limited and strictly prescribed way (see section 1.2.1).

If two processes in the set have the possibility of communicating in both directions with each other (this need not be the case for all pairs of processes), we say that they share a *link*. Hence we only consider bidirectional links. We will talk about *sending* and *receiving* messages over the link, although this link need not be a physical channel or transmission line. The set of processes together with their communication links can be represented by a *graph*, with processes as nodes or vertices and links as edges. As we only consider bidirectional communication links, this graph is an *undirected* graph. If we are mainly concerned with the graph representation of the distributed system, we will often use the terms *network* and (network) *nodes*.

An *event* in a distributed system is one of the following: (1) the sending of a message, (2) the receipt of a message, or (3) an internal event. Internal events can either be located at a process, such as the local execution of some program segment in a distributed algorithm, or can be located at a link, for example, the loss of a message in transit. We will model the latter type of internal event by means of processes also, so-called *link processes*. To avoid confusion, we will often use the term *processor* for a process that is considered to be located at a node in the network, although of course the correspondence between processes and processors need not be one to one.

In this book we exploit a layered structure more or less as provided by the OSI model. For each layer, we use the services provided by lower layers by making the

© Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
0521607558 - Protocols by Invariants
Anneke A. Schoone
Excerpt
More information

appropriate assumptions. For example, the communication protocol in the data link layer guards against the loss of messages. Thus in the network layer, we have the assumption that there is no loss of messages. Conversely, an assumption that is made in most chapters is that a link between two network nodes behaves like two FIFO queues of messages, one queue for each direction. This does not mean that we restrict ourselves to networks where this is actually the case, but we assume that this can be achieved by communication protocols present in the lower layers of the network, and that the question of how to achieve it is not our concern now. In the next section we give an overview of the typical network assumptions that are made throughout this book.

1.1.1.3 Network assumptions. There is a wide range of assumptions possible in distributed computing. For example, a network can be static or dynamic, nodes can be anonymous or have a unique identity, and communication can be reliable or unreliable. These different assumptions provide a way to focus attention on some aspects of distributed computing, while abstracting away from others, as they relate to the layered structure of a distributed system.

We can divide the network assumptions into those that pertain to the topology of the network, and those that pertain to the communication in the network. We will first discuss the assumptions related to the topology of the network.

We assume that a network consists of *processors* (nodes) connected by *undirected communication links.* Although directed networks are possible, we do not consider them. We assume that the processors have *unique identities.* As for the communication links, we consider two possibilities: either the links that are present remain present during the period considered and no new links arise, giving rise to a *static network*, or the links can come up and go down, giving rise to a so-called *dynamic network.* If two processors are incident to the same link, we call those processors *neighbors.* Unless otherwise stated, we assume that processors know the identity of their neighbors. In the way we formulate distributed algorithms or protocols we assume that there is only one link between two neighbors, but this is not really necessary. We also assume that the network contains no self-loops, i.e., links from a processor to itself.

Next there is the possibility of *processor failures.* Possible assumptions include: fault-free processors; processors that may crash and recover again (transient failures); processors that can crash, but are not allowed any activity any more after that (fail-stop); and *Byzantine* processors, making malicious faults. We model the failure of a processor (except in the last chapter) by the simultaneous failure of all its incident links, and the coming up of a processor by the coming up of some but not necessarily all of its incident links. We will sometimes need the assumption

© Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
0521607558 - Protocols by Invariants
Anneke A. Schoone
Excerpt
More information

that a processor failure does not include the loss of all memory of that processor, i.e., the processor can have access to stable memory, on some non-volatile storage medium.

Usually we assume that the network we consider is *connected*, but this is not always necessary. An assumption that is sometimes made is that every processor knows (an upper bound on) the number of processors in the network. A quantity like this can be used for example to limit the generation of messages or work (see section 3.2) in the system. Another parameter of the network that is useful in this respect is (an upper bound on) the *diameter* of the network. This can be used to kill messages after they have traveled over a sufficient number of links. It can be used even for dynamic networks, if there is an upper bound on the number of links that can go down, because there is a relation between the maximal diameter increase of a network and the number of links that go down [SBvL87].

In distributed computing, there are numerous algorithms designed for special network topologies, but we will only consider the general case. Sometimes we will discuss only the very simple topology of two processors connected by a link, if we study protocols for communication between two processors and assume that addressing is done correctly (in chapters 2 and 4). Then it is not necessary to consider the other parts of the network.

Processors can send messages to their neighbors over links, and processors can receive messages from their neighbors. However, messages cannot be received before they are sent. For this type of communication by *message passing*, there are two frequently used models. One is based on the idea of a handshake or *rendezvous*: if two processors want to communicate, the sender is blocked until the receiver is ready to receive. Thus the sending and the receiving action are *synchronous*: they take place at the same time. This model is used in programming languages like CSP [Hoa78], ADA, and OCCAM.

In the other, *asynchronous* model of message passing the sender is not blocked and it can take arbitrarily long to receive a message after it has been sent. In our terminology, it can take arbitrarily long for the message to arrive at its destination and be received after that (the receiver might be blocked until the message has arrived). In this model, processors need buffers to retain arrived messages until they can be processed. It is the model we will be using most in this book. A third model that is possible is to assume that messages arrive within bounded time.

Depending on the circumstances, we make different assumptions about the possible *communication errors*, i.e., the failures in message passing. These include *loss* of messages, for example due to a link failure, *garbled* messages, i.e., messages whose contents were changed during the transmission, and *duplication* of messages. We assume that a message cannot originate spontaneously on a link. As for garbled

© Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
0521607558 - Protocols by Invariants
Anneke A. Schoone
Excerpt
More information

messages, we usually assume (except in section 2.3) that a garbled message can be recognized as such, and thus can be purged and considered "lost". Duplication of messages can occur if messages can be routed by different routes, due to link failures. Routing by different routes can also cause the *reordering* of messages, i.e., that messages arrive at their destination in a different order than the order in which they were sent. Finally, if an assumption about the network is that messages arrive within bounded time, one can distiguish so-called *timing failures* if in fact they do not arrive in time.

1.1.2 Modeling distributed computations

For an appraisal of the general features in distributed computing, we start by considering the essential building blocks and only use complete protocols as illustrations. The building blocks are basically the separate statements of the actions that must be carried out in a distributed system, without specifying when and where they should take place. We can assemble bigger blocks by adding restrictions, e.g. in the control flow or the communication. We call such an assembled block an *operation*, and a set of statements or operations a *protocol skeleton*. If we assemble the statements or operations of a protocol skeleton into larger operations, we call the resulting protocol skeleton a *refined* protocol skeleton. Successive refinements can yield a complete protocol, while different successive refinements may yield another protocol for the same problem. Thus a protocol skeleton stands for a class of protocols, all of which can be obtained from this same skeleton by some refinement.

In denoting protocols, we use a mixture of English and an imperative programming language. As it is not necessary for understanding, we will not discuss things like parameter passing or give the syntax of the programming language.

1.1.2.1 Protocol skeletons. As said above, a protocol skeleton consists of a number of operations, each of which consists of a piece of (sequential) program. Operations can be carried out any number of times, by any processor, and at any time, and thus in any order. We assume that only one operation is carried out at a time. An operation is viewed as an atomic action, i.e., it cannot be interrupted. We do not specify anything about an assumed order in which the operations may take place, but an operation can contain a so-called *guard*: a boolean expression between braces { }. An operation may only be executed if it is *enabled*, i.e., if its guard is true, otherwise the operation is called *disabled* and nothing happens. For example, a processor can only execute the code for receiving a message if there is indeed a message present to receive.

© Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
0521607558 - Protocols by Invariants
Anneke A. Schoone
Excerpt
More information

The method we use for the correctness proofs of distributed algorithms is that of *assertional verification by system-wide invariants.* The idea is that if an assertion (a relation between process variables for example) holds initially, and is preserved by all possible operations, then it will hold always in the distributed system, whatever order of operations takes place in an actual execution of the distributed algorithm. Such an assertion is called a system-wide invariant. The advantage of the use of protocol skeletons in assertional verification is that refinements preserve invariants. If one has a refined protocol skeleton which can be viewed as a special instance of a protocol skeleton, then any system-wide invariant which holds for the protocol skeleton will hold also for the refined protocol skeleton. This is the case simply because the invariant was proven correct for any order of operations in the general case, and hence also for the special order of operations which will take place in the refined protocol skeleton.

The basic statements, and thus basic operations, in a distributed program for a processor i are: send a message to j (S_i), receive a message from j (R_i), and do an internal (local) computation (I_i), where j is any processor connected to i by a link. Operations and variables are subscripted by the identity of the processor that performs and maintains them, respectively. Usually, we only consider uniform distributed computations, in which all processors cooperate and contribute in the same way to the computation. Hence it is almost always the case that all processors have the same operations to execute, on their own local variables. Thus, if we give a protocol skeleton for processor i, we implicitly mean that the protocol skeleton contains those operations for *all* processors i. Thus we can give the basic protocol skeleton as follows.

Basic protocol skeleton:

S_i: **begin** send a message to some neighbor j **end**

R_i: {a message has arrived from j} **begin** receive the message from j **end**

I_i: **begin** compute **end**

As it is usually necessary to specify what the initial values of the processor variables are, we will do so for the variables used directly before the code of the operations, after the keyword **Initially**.

Apart from the operations that must be carried out at the processors, a protocol skeleton may contain operations pertaining to *link processes.* Link processes were described in section 1.1.1.2 and will be considered further in section 1.2.1. The name of a "link operation" is subscripted by the names of the processors incident

© Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
0521607558 - Protocols by Invariants
Anneke A. Schoone
Excerpt
More information

to the link. We also assume that the protocol skeleton contains that operation for *every* link in the network.

We can use the operations as *building blocks* for larger operations that we also consider as atomic. One might ask under what circumstances we are allowed to view operations as atomic. Lamport discussed this in general [Lam82], and for the case of message passing [Lam90]. Surprisingly, under some conditions one can even view the sending of a message and its subsequent delivery at the receiver in an asynchronous environment as one atomic action. In chapter 5 another approach is taken. There we discuss a class of protocols (atomic commitment protocols) that ensure that a set of operations is always executed as if it were one atomic action.

If we combine operations into larger ones, we add extra structure to the order of communication and/or the order of computation. This can be done in different ways, and yields different protocol skeletons. Some ways are more or less standard, and the resulting protocol skeletons will be referred to as *modes of computation*. Given a mode of computation, an idea of what information a message should contain, and a way to compute the wanted information from the received information, we have a general framework for a protocol.

We distinguish the following four modes of computation: *phasing, message-driven* computation, *simulated synchronous* computation, and *synchronous* computation. We will now describe each mode for fault-free static networks. Extensions to cope with possible errors and failures would obscure the basic ideas at this point. If errors may occur, we have to add the appropriate error operations (see section 1.2.1.2), and extend the basic protocol skeletons for these modes of computation with some means for coping with the errors. We will refine some of these protocol skeletons further to arrive at protocols for specific purposes. In chapter 3, we investigate the interrelation between the distributed computation per se and the mode of computation used, for the case of computing minimum-hop distances.

1.1.2.2 Phasing. The idea of phasing is to divide all the work to be done in a computation over different phases, and to allow a processor to begin working on the next phase only if all the work of the current phase is completed. Thus phasing adds some structure to the order of events in a computation, which was totally arbitrary until now. We add a variable for the current phase at each processor i: $phase_i$, and a new operation is added for the transition to the next phase: P_i. The most general protocol skeleton P which makes use of phasing is as follows.

Protocol skeleton P:

Initially $\forall\, i$: $phase_i = 0$.

© Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
0521607558 - Protocols by Invariants
Anneke A. Schoone
Excerpt
More information

S_i^P: **begin** send a message belonging to phase $phase_i$ to some neighbor k **end**

R_i^P: {a message has arrived from j}
begin receive the message from j and record it; compute **end**

P_i^P: {all messages of phase $phase_i$ have been received from all neighbors}
begin $phase_i := phase_i + 1$; compute **end**

Note that the internal computation operation I_i is now divided over operation R_i^P where the computation pertaining to the received message is done, and the operation P_i^P, the internal computation which effectuates the phase transition. Comparison of the operations S_j^P and R_i^P reveals a problem introduced by phasing. As in operation S_j^P a message belonging to the current phase $phase_j$ is sent, we can define the phase number of a message as the value of $phase_j$ at the moment that S_j^P was performed. However, in the corresponding receive operation R_i^P, nothing is mentioned about a phase number of a message. Ideally, the messages processor i receives while $phase_i = p$ would be the messages that i's neighbors sent while their phase number had the same value p. Now the problem is, what does processor i do when a message belonging to a different phase arrives: may it be received? There are several possible ways to deal with this problem.

First, it might be the case that the assumptions about communication on the links combined with the properties of a more specific protocol skeleton suffice to prove that the arrival of messages of the wrong phase cannot happen. Secondly, we could just refuse to receive the message if it belongs to a different phase, and add a guard to that effect to the operation R_i^P. We should take care not to introduce the possibility of deadlock then. Thirdly, the message could be received but ignored, as if it had been lost. The fourth possibility is, in case the message belongs to a later phase, to buffer it until the processor reaches the right phase. The fifth possibility is to just let the processor receive the message and perform the appropriate computation. It will depend on the circumstances what choice we make.

Another problem is the guard of operation P_i^P: "all messages of phase $phase_i$ have been received". There must be some way for processor i to evaluate this guard. One possibility, if the number of messages per phase is not fixed, is to include the number of messages to expect in a phase in the first message belonging to a phase, or to somehow mark the last message belonging to a phase (in the case of FIFO links).

1.1.2.3 Message-driven computation. The added structure in the order of computation in this case is that messages may only be sent upon receipt of another

© Cambridge University Press www.cambridge.org