

Chapter 2

Distributed Computing

Abstract Distributed computing studies the theory and methods to solve computational problems in distributed systems. There are many interesting and important problems that can be solved efficiently in distributed systems, such as data gathering in wireless sensor networks, computing graph properties, and leader election in a distributed system. In this chapter, we introduce the elementary concepts about distributed computing and some of the important components of distributed computing. In Sect. 2.1, we introduce the concept of distributed computing and present an example to illustrate it. Then, we present the communication models that are commonly utilized in Sect. 2.2, show the incompleteness of information in Sect. 2.3, and discuss the aspect of timing which plays an important role in distributed computing in Sect. 2.4.

2.1 What is Distributed Computing?

Distributed computing studies how to solve computational problems in distributed systems or environments. Generally speaking, an entity in a distributed system can compute its tasks completely locally against its “individual” goals. An analogy would be that each individual has his own will and sentiments about certain public events. However, these entities can also cooperate to solve global computational problems which are tough or impossible for a single entity to handle, even though they may not know the others’ information. For example, all individuals can contribute their strength in crowdsourcing to provide some common needed services or to achieve some common goal [1, 2, 6].

We use a simple example of wireless sensor networks to explain distributed computing. As illustrated in Fig. 2.1, 9 sensors are deployed in the environment to detect the temperature. The goal is to find out the highest temperature in the area at the base station through the sensors. As shown in the figure, the sensors form a wireless network where any two sensors are supposed to be connected if they are within each other’s range of communication. Each sensor node can detect the temperature locally and then share this local data by communicating with nearby neighboring sensors. The 9 sensors can sense different temperature data and after receiving the others’ data

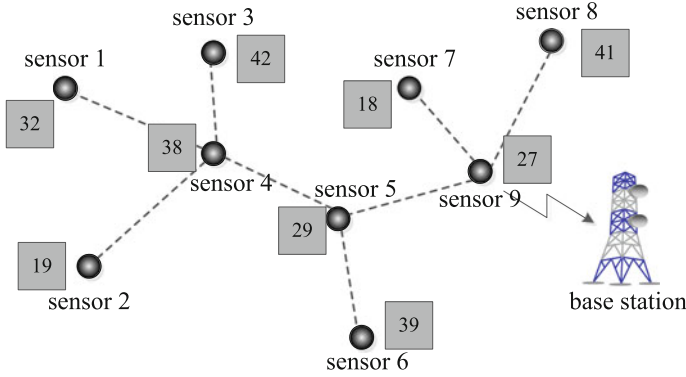


Fig. 2.1 An example of computing the maximum temperature in a wireless sensor network

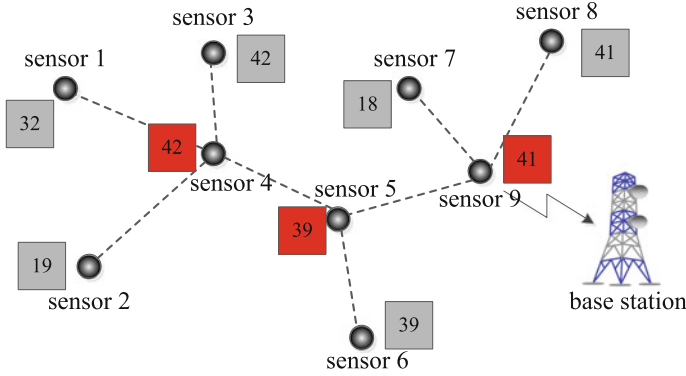


Fig. 2.2 An example of update dating when we are to compute the maximum temperature through the wireless sensor network

(the temperature data), each sensor can compute which temperature is the highest among all neighbors and then update the data.

In Fig. 2.2, sensor 4 will update the local highest temperature data to 42 since it can collect four values {19, 31, 38, 42}. Similarly, sensor 5 will update the local temperature to 39 and sensor 9 to 41. However, having updated its data, a sensor would communicate its updated data with the neighbors again. For example, when sensor 4 has updated its max value to 42, it will also send the data to sensor 5, which would also update the local highest temperature data to 42. After sensor 5 has updated to 42, it will then send the information to sensor 9 which will finally send the data to the base station. In this fashion, all sensors can cooperate to compute the highest temperature and inform the base station.

Distributed computing is quite different from centralized computing where all entities know the global information ahead of time. Considering the above example, if all nodes know the others' temperature data in advance, they (including sensor 9)

can easily find out the global highest temperature. In distributed computing, although each entity focuses on its private computation tasks, through reasonable communication (data transmission), the private computations of the entities can be combined to realize a global computational task, which should produce the same result as centralized computing, but with the extra cost of longer elapsed time.

Distributed computing has the following three traits. The first one is the communication model, which dictates how data are to be transferred between two entities; the second one is the incompleteness of information, which reveals to what extent an entity can know about the whole system; the third one is synchrony and timing, which indicates when the entity starts its computation and how the computation should be paced.

2.2 Communication Model

In a distributed system, two connected entities can exchange and share their information. There are two commonly used communication models:

- (1) *Message passing model (MPM)*: an entity can send data to its neighbors through the connecting edges;
- (2) *shared memory model (SMM)*: the entities can use some kind of common memory to perform data transmission.

In the message passing model, the communications between two entities are explicit. When one entity wants to share information, it can transmit the data through the edges connecting the neighboring entities. A large number of message passing models have been proposed in the past; here, we describe two of them.

One type of message passing model is called *point-to-point* communication, which allows direct information transmission between a specific pair of entities. For example, in Fig. 2.3, the edges with both start and end arrows represent bidirectional communication connections (or channels) between two entities, where bidirectional communication means one node (entity) can both send and receive information to/from the other node of the edge. In the figure, nodes *a* and *b* can both share information with the other (the red rectangle represents the message). However, some systems may use just unidirectional communication connections, i.e. one node can only send or receive from the other node. We use an edge with only one arrow in the figure to represent a unidirectional connection. As shown in the figure, node *c* has a directed edge to node *a*, which means node *c* can send its message to node *a*, but node *a* cannot share its information with node *c*. Edges from node *d* to *c*, node *e* to node *d*, node *e* to node *g*, and node *h* to node *f* are likewise unidirectional edges.

From the figure, node *a* can get node *b*'s and node *c*'s data through the edges, but it cannot receive both pieces of data at the same time. Point-to-point communication means that the data transmission process happens between a pair of neighbors, but one node cannot receive data from multiple senders simultaneously. Through continuous communication, node *b* can also receive node *c*'s information through node *a*, but

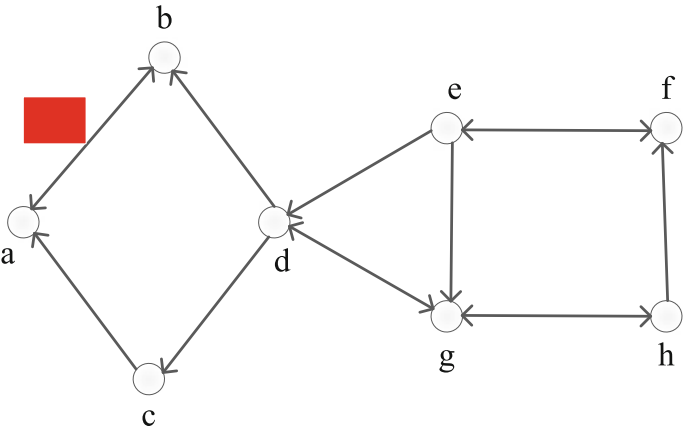


Fig. 2.3 An example of message passing model

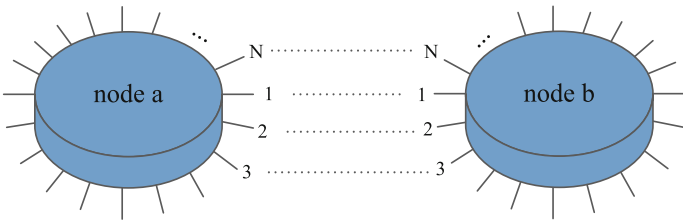


Fig. 2.4 An illustration of message passing model with external ports for connection

it takes longer time and we say the information is delayed. However, node *c* cannot get node *b*'s information through the depicted topology.

Another type of message passing model is called *broadcast*, where one node can send its information to *all* the neighbors simultaneously. Its main difference from the point-to-point communication model, where a message can be only sent to one receiver at a time, is that it allows multiple recipients at the same time. For example, in Fig. 2.3, node *d* can broadcast its message and nodes *b*, *c*, *g* can receive this message simultaneously. This broadcast model has many important applications, such as flooding in network construction [4, 5], and fast message propagation [7].

The first type of communication model is widely adopted since it reveals the connection patterns directly. However, there is a more refined model of communication, as shown in Fig. 2.4. The edges in the graph (Fig. 2.4) represent direct connections between two nodes, which also means the two nodes are relatively close to each other; but they still need to rely on a connection channel for communication. Suppose each node has a number of *ports*, i.e. external connection points, and every communication channel connects two ports of two connected nodes. When node *a* tries to send a message to node *b*, it should load the data onto an appropriate port, such as port 1; when the message arrives at node *b*, it will be stored in *b*'s local buffer. However,

if node b does not choose port 1 or the connection channel between the two ports cannot be established, node b cannot receive the message. This book focuses on the rendezvous problem in distributed systems, which is the process of establishing a common communication link (connection) between two connected external ports of the communicating entities.

2.3 Information Incompleteness

In executing computational tasks in a networked system, centralization helps solve the problems from a global view where each node in the system has full knowledge of all relevant information about the tasks. Therefore, each node will achieve the same result with the full knowledge (we do not consider randomized algorithms here, which may lead to different results even with same input). The completeness of information is equivalent to a full input to any problem in such a centralized setting.

However, centralized setting is hard to implement in real distributed systems. For example, there are more and more mobile phones becoming active nowadays, and it is costly to construct a static network and inform all users about the new phones' information and the constructed topology. In distributed systems, every entity needs to cope with the fact that only partial information of the system is available. This is equivalent to the situation where only partial input to a problem is available, and the user has to execute its task with its stored or information obtainable from the surrounding. Moreover, each entity may not even be aware of who the other participating entities are, where the computation begins, and which stage of the computation the others are currently at. These uncertainties lead to difficulties in coordinating the joint computation of the entities of a common task.

In some practical applications or computational tasks, the entity of a distributed system may not need to have the full knowledge about the system. For example, if each entity should compute the number of connected neighbors dynamically (since some entities may join in or leave the system at any time), it only needs to find out the active entities within its communication range. Actually, full system information does not help perform such a task. Therefore, the entity may not need to know all the outsiders' information, and collectively the entities can also work well with only local information in solving many computational problems.

There are a variety of models that govern concern the topological knowledge. One typical model is known as an anonymous system, where all entities (or nodes) are indistinguishable and they have no identification labels. Moreover, each node knows nothing about the topology of the network. This model is at the extreme end of the spectrum, which makes distributed computing difficult. For example, it is hard for a node to find out whether it has sent a message to all neighbors in a point-to-point communication model, since it cannot tell whether two "different" end-points are the same node. A more realistic model assumes that each node is assigned a unique identifier and the node knows the identities of its neighbors. For example, a computer or a mobile phone has a unique MAC address, which can be discovered by others.

Some models assume a node know even more about the system. For example, each node may know the k -hop network topology, which means it is able to find out the nodes that are reachable within k hops. This subnetwork information can help solve some problems to a certain extent. When k becomes larger, more information can be obtained by each node. The most powerful model assumes each node can have the complete topological knowledge of the system, which degenerates to the centralized setting; centralized setting is impractical and hard to implement.

2.4 Timing and Synchrony

In distributed computing, timing is a subtle concept that deals with when the entities may execute their tasks or computational steps [3]. In our normal living, we have a global clock that tells the time and all the entire world agrees to the same rules for defining time. However, in a distributed system, timing is hard to coordinate and a global clock is hard to implement if it is to be used by all distributed entities.

We mention two models that have to do with timing: synchronous model and asynchronous model.

In the *synchronous model*, all entities in the distributed system share a global clock, which indicates the exact times of the events in the system. Time can be considered as divided into slots of equal length (the analogy is that we use “second” as the elementary unit in physical clocks), and each entity should execute the following three steps in each slot:

- (1) Receive messages from (some of) the neighbors;
- (2) execute local computation based on its local status and the received messages;
- (3) send messages to (some of) the neighbors.

The time cost of local computation on each entity is assumed to negligible compared to the message transmissions. Therefore, in the model, an entity only needs to wait for its message and then send out a computed message. This model satisfies an important property: if entity a sends a message to entity b in slot t , the message must be received by entity b before or in slot $t + 1$. Thus, all entities’ activities can be regarded as driven by a global clock.

However, in the *asynchronous model*, messages are not guaranteed to be transmitted to the other entity timely. All entities do not access the global clock and they have to decide on their own actions. Generally speaking, messages sent from one entity to another will arrive within some finite but unpredictable time. Therefore, one cannot rely on the elapsed time to deduce whether a message was sent from a neighbor or not. Thus, the algorithms for this model are always event driven, i.e. the entity will execute its local computations when a message is received, or when some local memory has changed. Therefore, the execution steps are as follows:

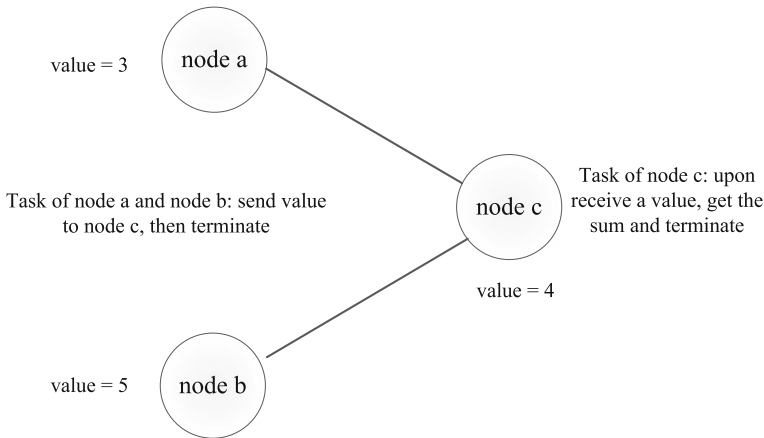


Fig. 2.5 An example of distributed computing upon received messages

- (1) Wait for an event, where the event could be receiving messages from (some of) the neighbors, or the local memory has changed;
- (2) execute local computations based on its local memory and the received messages;
- (3) trigger an event, such as sending messages to (some of) the neighbors, or change the local memory.

Clearly, the entities' computations could be affected by the messages' arrival times. However, it is impossible to rely on the ordering of the arrived messages for executing local computations.

For example, in Fig. 2.5, node *a* and node *b* are connected to node *c*. The tasks for node *a* and node *b* are to send its local value to node *c*, and then terminate. Meanwhile, the task of node *c* is to add the received values and update its local value; and then, it will terminate. In the asynchronous model, the messages could be delayed by different reasons. Suppose node *a* sends the value to node *c* earlier than node *b*, if the messages can arrive at node *c* timely and sequentially, node *c* will get node *a*'s value and update its local value to $3 + 4 = 7$. However, in the asynchronous system, node *b*'s message may arrive earlier at node *c* and it will update the value to $5 + 4 = 9$, which leads to different results.

Therefore, timing plays an important role in distributed computing. Some asynchronous models assume the entities start the algorithm in different time slots, but the messages are guaranteed to be received in the next time slot. Therefore, in the above figure, if one node sends its local value earlier than the other, for example, node *a* sends the data $\Delta > 0$ time slots earlier than node *b*, node *c* will update the value to $3 + 4 = 7$ timely, and then terminate. In this book, we design efficient distributed algorithms for both synchronous and asynchronous scenarios; we use exactly this type of asynchronous model which we will introduce later.

References

1. Doan, A., Ramakrishnan, R., & Halevy, A. Y. (2011). Crowdsourcing systems on the world-wide web. *Communications of the ACM* 54(4).
2. Huberman, B. A., Romero, D. M. & Wu, F. (2009). Crowdsourcing, attention and productivity. *Journal of Information Science*, 35(6).
3. Lamport, L.(1978). Time, clocks, and the ordering of events in a distributed system. *Operating System*.
4. Lim, H. & Kim, C. (2001). Flooding in wireless ad hoc networks. *Computer Networks*, 24(3–4).
5. Liu, H., Jia, X., Wan, P.-J., Liu, X., & Yao, F. F. (2001). A distributed and efficient flooding scheme using 1-hop information in mobile ad hoc networks. *IEEE Transactions on Parallel and Distributed Systems*, 18(5).
6. Morschheuser, B., Hamari, J., Koivisto, J. (2016). Gamification in crowdsourcing: A Review. In *49th Annual Hawaii International Conference on System Sciences (HICSS)*
7. Ye, S., Wu, S. F. (2010). Measuring message propagation and social influence on twitter. In *International Conference on Social Informatics*, Springer, Berlin

Rendezvous in Distributed Systems

Theory, Algorithms and Applications

Gu, Z.; Wang, Y.; Hua, Q.-S.; Lau, F.

2017, XIX, 262 p. 99 illus., 22 illus. in color., Hardcover

ISBN: 978-981-10-3679-8