In this chapter we discuss the abstractions that are commonly assumed to be provided by a distributed system, and discuss how these abstractions can be supported. We consider each of these abstractions a "building block," upon which fault tolerant applications can be constructed. For each of the "building blocks" we first specify the problem, and then discuss some methods to implement a system from existing components (that typically do not provide the desired property) that support the property of the "building block."

3.1 Byzantine Agreement

Often it is assumed that when a component in a computer system fails, it behaves in a certain well-defined manner, though its behavior may be different from its failure-free behavior. Examples of these types of failures are a component always showing a 0 at the output (stuck at 0), or simply stopping execution. However, in the most general case, when a component or a system fails, its behavior can be totally arbitrary. In particular, the faulty component/system may send totally different information to the different components with which it communicates. With such types of failures, reaching agreement between different components is quite complex. The problem of reaching agreement in a system where components can fail in an arbitrary manner is called the *Byzantine generals problem*. As defined earlier, the failure mode in which a component behaves totally arbitrarily and may even send different information to different components is called a Byzantine failure. We will refer to protocols that are used to reach agreement with Byzantine failures as the *Byzantine agreement protocols*.

Byzantine agreement is an important problem, since the failure mode considered in this problem is most general, and if we can handle it satisfactorily, we can be sure that most types of failures can be masked. Also, real computer systems/components often behave arbitrarily when they fail. Hence, if we want the components to fail in a certain manner, we have to construct such a system from systems that can fail in any manner. In other words, if we want a component with a "nice" failure mode on which a fault tolerant system can be built to mask the failure, we have to build it from ordinary components that may fail in an arbitrary manner. We will see later how a solution to the Byzantine generals problem is used to support fail stop processors, which are frequently assumed in protocols for fault tolerance in distributed systems. A Byzantine agreement protocol is therefore a building block for other building blocks.

3.1.1 Problem Definition and Impossibility Results

The Byzantine generals problem needs a careful definition. We will consider a system with many components in which components exchange information with each other. We will consider a distributed system in which nodes are the components and information is exchanged by message passing. The components (nodes) may be faulty and may exhibit Byzantine failures. That is, a faulty node may send different values to different nodes (for the same data).

The basic goal to be achieved is to obtain a consensus among all nonfaulty nodes in this context. Each node has to make a decision based on values it gets from the other nodes in the system. We require all nonfaulty nodes to make the same decision. Hence, the goal is to ensure that all nonfaulty nodes get the same set of values. If all nonfaulty nodes get the same set of values from different nodes, consensus can easily be achieved by all nodes using the same procedure for making the decision. The problem is complicated by the possibility that a faulty node may send different values to different nodes. So a simple message exchange method for transmitting values of different nodes will not work (this can suffice in a failure-free environment, or in some situations with simpler failure modes).

To ensure that each nonfaulty node receives the same set of values, we can state a simpler requirement that is equivalent. The requirement is that every nonfaulty node in the system uses the same value for a node i for decision making. Clearly, if this property is satisfied for all nodes, we can say that the set of values at each nonfaulty node is the same. Hence, the general problem of consensus is reduced to agreement by nodes in a system on the value for a particular node. This solution can then be used to disseminate values of all the nodes in the systems. This is stated formally as the following two requirements [LSP82]:

1. All nonfaulty nodes use the same value $v(i)$ for a node i.

2. If the sending node i is nonfaulty, then every nonfaulty node uses the value i sends.

This problem is also called the *interactive consistency* problem. Note that this formulation allows any type of behavior during failure, and that the sending node (node i) may itself be faulty. Any solution to this problem should take this into account. In particular, a solution to this problem must consider the possibility of a faulty node acting "maliciously" and sending messages that will thwart the consensus process.

Given a protocol to solve this problem, the protocol can be used by each node to send its value to other nodes. Then each node can use the same procedure to take

its decision based on the values obtained from the different nodes, thereby achieving consensus.

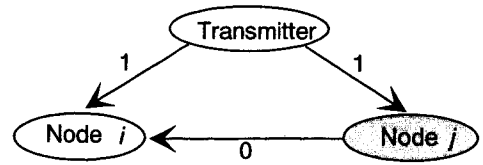
The Byzantine agreement problem is hard because the information sent by a node to another cannot be trusted. Hence, to agree on a value sent by a node, besides getting the value from that node, the value as received by other nodes is also needed to verify the original value. The problem becomes complex, since these "forwarding" nodes may also be faulty (and may behave arbitrarily or maliciously). The problem can be solved only if the number of faulty nodes in the system is limited.

If the distributed system is asynchronous (i.e., there is no bound on the relative speeds of nodes or the communication delays), then it can be shown [FLP85] that agreement is impossible if even one processor can fail, and even if the failure is a crash failure, which is much more benign than Byzantine failure. This is because in such systems the failure of a node to send a message (because it is faulty) cannot be distinguished from the situation where the node and the communication network are extremely slow. In such a situation, a node can never detect the absence of a message with certainty. Consequently, a faulty node can block the consensus algorithm merely by not sending a message. Hence, most of the algorithms proposed for solving the Byzantine generals problem are for synchronous distributed systems where the message delays and the differences in the relative speeds of processors are bounded.

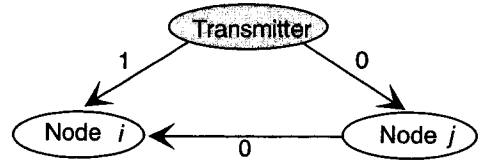
For the rest of this section we assume that the distributed system is synchronous. Let us first see that in a three node system, if one node is faulty, this problem cannot be solved if ordinary message passing is employed by nodes [LSP82]. Suppose the nodes need to agree to a Boolean value: true (1) or false (0). Let us consider the two scenarios shown in Fig. 3.1 with one of the three nodes as faulty [LSP82].

In the first scenario, one of the receiving nodes j is faulty, and though the sender sends it a 1, it transmits to node i that a 0 was sent by the sender. In the second scenario, the sending node itself is faulty, and it sends a 1 to node i and a 0 to node j, which the node j faithfully forwards to i. Both these situations are indistinguishable to node i, and it cannot decide whether the value sent by the sender should be considered as correct, or the value forwarded by node j. If it decides to accept the value sent by the sender (i.e., 1), then in the same scenario node j will accept the value 0. This will violate the second requirement.

It has been shown [LSP82, PSL80, FLM86] that with ordinary messages it is impossible to solve this problem unless more than two-thirds of the nodes are nonfaulty. That is, the number of faulty nodes has to be less than one-third of the total number of nodes. To cope with m faulty nodes, at least $3m + 1$ nodes are needed to reach consensus.



Scenario 1: Node j is faulty



Scenario 2: Transmitter is faulty

Figure 3.1: Two scenarios

The problem is simplified if the messages can be "signed." That is, a node can attach a "signature" to each message it sends. The message with a signature is such that no node can tamper with the message contents of a nonfaulty node without the alteration being detected by other nonfaulty nodes. Hence, when a nonfaulty sender sends a message to other nodes, a faulty node cannot tamper with its message and forward it to other nodes. If it attempts to do that, the tampering will be detected. In this situation, an arbitrary number of faulty nodes can be tolerated [LSP82]

3.1.2 Protocol with Ordinary Messages

Let us now discuss a protocol for solving the Byzantine generals problem in a distributed system, where ordinary messages are employed. As mentioned above, consensus can be reached in the presence of m faulty nodes only if the total of the nodes is at least $3m + 1$. We assume that a fault-free node executes the protocol correctly; a faulty node can behave in any manner. First, let us define more precisely the assumptions we are making about the message passing system [LSP82].

A1. Every message that is sent by a node is delivered correctly by the message system to the receiver.

A2. The receiver of a message knows which node has sent the message.

A3. The absence of a message can be detected.

Assumption A1 ensures that a faulty node cannot interfere with communication

between other nodes. A2 ensures that a faulty node cannot masquerade as another node. This assumption has some implications for the underlying network. It essentially means that there is a dedicated physical channel between two nodes, and no message switching is used. With switching, an intermediate faulty node in the path can corrupt the message. The practical implication of these two assumptions is that the physical communication network must be fully connected, i.e., there must be a direct physical line between any pair of nodes. A3 ensures that a faulty node cannot foil the consensus attempt by simply not sending messages as required by the protocol. This assumption is typically implemented using timeouts, and requires clocks at different nodes to work at the same rate. It also implies that the message delays and the differences in the relative speeds of processors are bounded.

The algorithm presented here will work only if the message passing system satisfies the assumptions A1 through A3. It was proposed by Lamport, Shostak, and Pease in [LSP82]. This algorithm will be referred to as *interactive consistency algorithm, ICA(m)*, for all nonnegative integers m. The integer m represents the number of faulty nodes. Let n represent the total number of nodes. For this algorithm to achieve consensus, we must have $n \geq 3m + 1$. One node is designated as the *transmitter*, whose value has to be agreed upon by other nodes. Other nodes are *receivers*. If a node does not send a message it is supposed to send (recall that our assumption ensures that this will be detected by a receiver), the receiver node uses a *default value*. We take this default value to be 0. The algorithm is specified inductively, and is given in Fig. 3.2 [LSP82].

This algorithm works in rounds, each round consisting of message exchanges between nodes. In round 1, the transmitter sends values to other $n - 1$ nodes. The receiver nodes cannot trust the values they receive from the transmitter, since the transmitter may be faulty. Hence, a receiver node first determines from other nodes the value they received from the transmitter. A node takes the majority values it gets from other nodes and the value it got from the original transmitter. This majority value is taken to be the value sent by the transmitter.

In other words, each receiver has to communicate the value it receives from the transmitter in round 1 to others (or an arbitrary value, if it has failed) in round 2. In this round, each of the receivers acts as a transmitter, and sends messages to nodes other than the transmitter (and itself). That is, we can consider it as a system of $n - 1$ nodes, in which each node sends $n - 2$ messages. Messages sent by a node i in this round essentially aim to inform other nodes about the value it received from the transmitter. However, since a receiver cannot trust a transmitter in this round (since a node which is "forwarding" the message of the original transmitter may be faulty), by the same argument as used after round 1, now the version of values as received by other nodes is needed to determine the value sent by a node. For this, messages

Algorithm ICA(0).

1. The transmitter sends its value to all the other $n - 1$ nodes.
2. Each node uses the value it receives from the transmitter, or uses the default value, if it receives no value.

Algorithm ICA(m), $m > 0$.

1. The transmitter sends its value to all the other $n - 1$ nodes.
2. Let v_i be the value the node i receives from the transmitter, or else be the default value if it receives no value. Node i acts as the transmitter in algorithm $ICA(m - 1)$ to send the value v_i to each of the other $n - 2$ nodes.
3. For each node i, let v_j be the value received by the node j ($j \neq i$). Node i uses the value $majority(v_1, ..., v_{n-1})$.

Figure 3.2: Interactive consistency algorithm

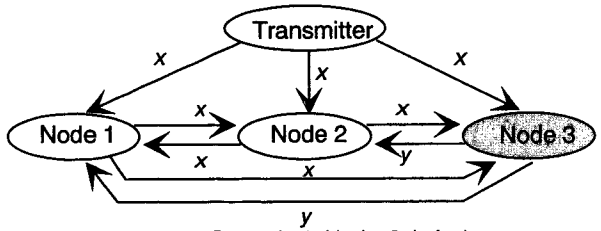
are sent in round 3. Once again, for a node i, a majority of the values (one received from the node i as a message in round 2, and $n - 3$ messages received from other nodes from round 3) is taken to determine the value sent by a node i. This value is taken as the value sent by the original transmitter to the node i. A majority of these values was taken after round 1 to determine the transmitter's value.

This distrust in round 2 requires round 3 of message exchange. In round 3, the system can be considered as consisting of $n - 2$ nodes, and each node will send, for each message it receives in round 2, its version to the remaining $n - 3$ nodes. Again, a majority is taken as the value. However, again due to the possibility of faulty nodes, messages of this round cannot be trusted, and hence the next round is needed to agree on the value sent on this round, which requires another round, and so on.

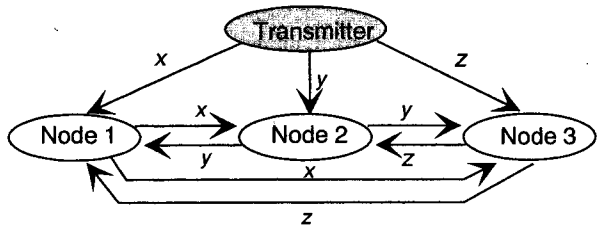
Eventually, in round $m + 1$, ICA(0) will be called. In this case, the recursion ends, and a node, for each message it gets in round m, simply transmits it (if the node is faulty, it may transmit any value). A majority of the values directly received in the messages is taken to be the value sent by a node in round m. This value then percolates up the recursion chain.

To better understand the protocol, consider a system with 1 faulty node and a total of 4 nodes [LSP82]. First consider the case where the transmitter is reliable,

and one of the receivers (node 3) is faulty. In this case, the transmitter during its execution of $ICA(1)$ will send the same value (say x) to the three remaining nodes (one of which is faulty). During execution of $ICA(0)$, the two reliable nodes will forward the value they received from the transmitter (x) faithfully to other nodes. However, the faulty node can send any value to others. Suppose it sends y to nodes 1 and 2. In this case, node 2 (and node 1) will get the values (x, x, y). The majority of this is x. Hence, the value agreed to by node 2 is the same as the one sent by the reliable transmitter node. This situation for node 2 is shown in Scenario 1 of Fig. 3.3.



Scenario 1: Node 3 is faulty



Scenario 2: Transmitter is faulty

Figure 3.3: Algorithm $ICA(1)$

Now let us consider the case where the transmitter itself is faulty, and the receivers are reliable. This is shown in Scenario 2 of Fig. 3.3. Suppose the faulty transmitter sends x to node 1, y to node 2, and z to node 3 during $ICA(1)$. Since the nodes 1, 2, and 3 are reliable, during $ICA(0)$ they will transmit faithfully to others the value they received from the transmitter. Hence, each one of them will get the values (x, y, z). Since each executes the same $majority$ algorithm, they will agree on the same value, regardless of the actual values of x, y, and z, thereby satisfying the requirements.

Now let us try to understand intuitively why taking a majority works, and why this algorithm can only handle m faulty nodes in a system of $3n + 1$ nodes. A formal proof can be found in [LSP82]. For $ICA(m)$, in the last round, the system for this round of message exchanges consists of $(3m + 1) - m = 2m + 1$ nodes, as there are originally $3m + 1$ nodes, and in each new round, the original transmitter is not included in the system. In the worst case, all the faulty nodes are still in the system. That is, all the transmitters were reliable, and the faulty nodes are receivers in the mth round. The value sent by these nodes in the $m + 1$ round (with $ICA(0)$) will be used to determine the majority. Since there are, at most, m faulty nodes, and a total of $2m + 1$ nodes, a majority of nodes are fault-free. Since each fault-free node takes a majority, all the fault-free nodes will agree on the same value, since a majority of nodes are fault-free and a fault-free node faithfully "forwards" a message. If there were more faulty nodes, then a majority agreement may lead to different nodes agreeing on different values in round m (since faulty nodes may send a different value to different nodes), which will then affect the values in earlier rounds.

It is clear that with the protocol described by $ICA(m)$, different nodes in the system can reach an agreement on the value for the transmitter node, as long as there are no more than m nodes that are faulty (and there are a total of at least $3m + 1$ nodes). Clearly, this can be used by all the nodes to reach an agreement on their values. Hence, by executing $ICA(m)$ for each node, each node in the system will have the same value for every node. Clearly then, all the nonfaulty nodes will reach the same decision, if the decision is based on the values from the different nodes. In other words, a consensus can be reached by different nodes in a distributed system using an $ICA(m)$ protocol.

3.1.3 Protocol with Signed Messages

The protocol $ICA(m)$ is complicated because it has to handle the case that a faulty receiver node may "forward" a different value from the one received by it from the repeater. And a receiver of this message has no way of determining whether the sender node has tampered with the original message. The problem becomes easier if we restrict the ability of nodes to tamper with messages. This can be achieved by a transmitter sending a "signed" message. That is, a node adds a digital signature to its message. The signature is such that if any node changes the contents of the message it receives from the transmitter and then forwards the altered message, this change can be detected by the receiver. The signature can be obtained using encryption techniques.

With signed messages, the impossibility result of being able to tolerate m faults in a $3m + 1$ node system does not hold. Agreement can be reached for an arbitrary