focus

fault tolerance

Mastering Agreement Problems in Distributed Systems

Michel Raynal, *IRISA*

Mukesh Singhal, *Ohio State University*

Overcoming agreement problems in distributed systems is a primary challenge to systems designers. These authors focus on practical solutions for a well-known agreement problem—the nonblocking atomic commitment.

Distributed systems are difficult to design and implement because of the unpredictability of message transfer delays and process speeds (asynchrony) and of failures. Consequently, systems designers have to cope with these difficulties intrinsically associated with distributed systems. We present the *nonblocking atomic commitment* (NBAC) agreement problem as a case study in the context of both synchronous and asynchronous distributed systems where processes can fail by crashing. This problem lends itself to further exploration of fundamental mechanisms and concepts such as *time-out*, *multicast*, *consensus*, and *failure detector*. We will survey recent theoretical results of studies related to agreement problems and show how system engineers can use this information to gain deeper insight into the challenges they encounter.

Commitment protocols

In a distributed system, a transaction usually involves several sites as participants. At a transaction's end, its participants must enter a commitment protocol to commit it (when everything goes well) or abort it (when something goes wrong). Usually, this protocol obeys a two-phase pattern (the *two-phase commit*, or 2PC). In the first phase, each participant votes Yes or No. If for any reason (deadlock, a storage problem, a concurrency control conflict, and so on) a participant cannot locally commit the transaction, it votes No. A Yes vote means the participant commits to make local updates permanent if required. The second phase pronounces the order to commit the transaction if all participants voted Yes or to abort it if some participants voted No.

Of course, we must enrich this protocol sketch to take into account failures to correctly implement the failure atomicity property. The underlying idea is that a failed participant is considered to have voted No.

Atomic commitment protocols

Researchers have proposed several atomic commitment protocols and implemented them in various systems.[1,2] Unfortunately, some of them (such as the 2PC with a main coordinator) exhibit the *blocking* property in some failure scenarios.[3,4] "Blocking" means

0740-7459/01/$10.00 © 2001 IEEE

Authorized licensed use limited to: University of Minnesota. Downloaded on January 28, 2010 at 14:57 from IEEE Xplore. Restrictions apply.

The Two-Phase Commit Protocol Can Block

A coordinator-based two-phase commit protocol obeys the following message exchanges. First, the coordinator requests a vote from all transaction participants and waits for their answers. If all participants vote Yes, the coordinator returns the decision Commit; if even a single participant votes No or failed, the coordinator returns the decision Abort.

Consider the following failure scenario (see Figure A).[1] Participant p_1 is the coordinator and p_2, p_3, p_4, and p_5 are the other participants. The coordinator sends a request to all participants to get their votes. Suppose p_4 and p_5 answer Yes. According to the set of answers p_1 receives, it determines the decision value D (Commit or Abort) and sends it to p_2 and p_3, but crashes before sending it to p_4 and p_5. Moreover, p_2 and p_3 crash just after receiving D. So, participants p_4 and p_5 are blocked: they cannot know the decision value because they voted Yes and because p_1, p_2, and p_3 have crashed (p_4 and p_5 cannot communicate with one of them to get D). Participants p_4 and p_5 cannot force a decision value because the forced value could be different from D. So, p_4 and p_5 are blocked until one of the crashed participants recovers. That is why the basic two-phase commit protocol is blocking: situations exist where noncrashed participants cannot progress because of participant crash occurrences.

Reference

1. Ö. Babaoğlu and S. Toueg, "Non-Blocking Atomic Commitment," *Distributed Systems*, S. Mullender, ed., ACM Press, New York, 1993, pp. 147–166.

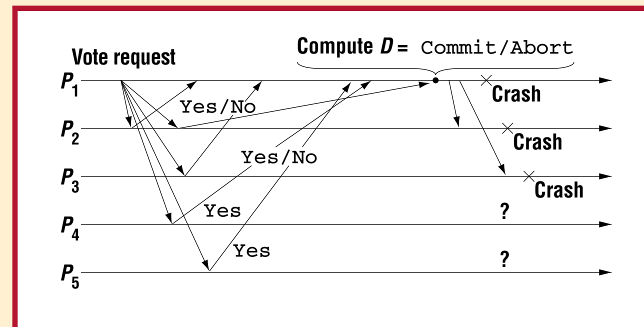


Figure A. A failure scenario for the two-phase commit protocol.

that nonfailed participants must wait for the recovery of failed participants to terminate their commit procedure. For instance, a commitment protocol is blocking if it admits executions in which nonfailed participants cannot decide. When such a situation occurs, nonfailed participants cannot release resources they acquired for exclusive use on the transaction's behalf (see the sidebar "The Two-Phase Commit Protocol Can Block"). This not only prevents the concerned transaction from terminating but also prevents other transactions from accessing locked data. So, it is highly desirable to devise NBAC protocols that ensure transactions will terminate (by committing or aborting) despite any failure scenario.

Several NBAC protocols (called 3PC protocols) have been designed and implemented. Basically, they add handling of failure scenarios to a 2PC-like protocol and use complex subprotocols that make them difficult to understand, program, prove, and test. Moreover, these protocols assume the underlying distributed system is synchronous (that is, the process-scheduling delays and message transfer delays are upper bounded, and the protocols know and use these bounds).

Liveness guarantees

Nonblocking protocols necessitate that the underlying system offers a *liveness guarantee*—a guarantee that failures will be eventually detected. Synchronous systems provide such a guarantee, thanks to the upper bounds on scheduling and message transfer delays (protocols use time-outs to safely detect failures in a bounded time). Asynchronous systems do not have such bounded delays, but we can compensate for this by equipping them with unreliable failure detectors, as we'll describe later in this article. (For more on the difference between synchronous and asynchronous distributed systems, see the related sidebar.) So, time-outs in synchronous systems and unreliable failure detectors in asynchronous systems constitute building blocks offering the liveness guarantee. With these blocks we can design solutions to the NBAC agreement problem.

Distributed systems and failures

A distributed system is composed of a finite set of sites interconnected through a communication network. Each site has a local memory and stable storage and executes one or more processes. To simplify, we assume that each site has only one process. Processes communicate and synchronize by exchanging messages through the underlying network's channels.

A synchronous distributed system is characterized by upper bounds on message transfer delays, process-scheduling delays, and message-processing time. δ denotes an upper time bound for "message transfer de-

Authorized licensed use limited to: University of Minnesota. Downloaded on January 28, 2010 at 14:57 from IEEE Xplore. Restrictions apply.

A Fundamental Difference between Synchronous and Asynchronous Distributed Systems

We consider a synchronous distributed system in which a channel connects each pair of processes and the only possible failure is the crash of processes. To simplify, assume that only communication takes time and that a process that receives an inquiry message responds to it immediately (in zero time). Moreover, let Δ be the upper bound of the round-trip communication delay. In this context, a process p_i can easily determine the crash of another process p_i. Process p_i sets a timer's time-out period to Δ and sends an inquiry message to p_i. If it receives an answer before the timer expires, it can safely conclude that p_i had not crashed before receiving the inquiry message (see Figure B1). If p_i has not received an answer from p_i when the timer expires, it can safely conclude that p_i has crashed (see Figure B2).

In an asynchronous system, processes can use timers but cannot rely on them, irrespective of the time-out period. This is because message transfer delays in asynchronous distributed systems have no upper bound. If p_i uses a timer (with some time-out period Δ) to detect the crash of p_i, the two previous scenarios (Figures B1 and B2) can occur. However, a third scenario can also occur (see Figure B3): the timer expires while the answer is on its way to p_i. This is because Δ is not a correct upper bound for a round-trip delay.

Only the first and second scenarios can occur in a synchronous distributed system. All three scenarios can occur in an asynchronous distributed system. Moreover, in such a system, p_i cannot distinguish the second and third scenarios when its timer expires. When considering systems with process crash failures, this is the fundamental difference between a synchronous and an asynchronous distributed system.

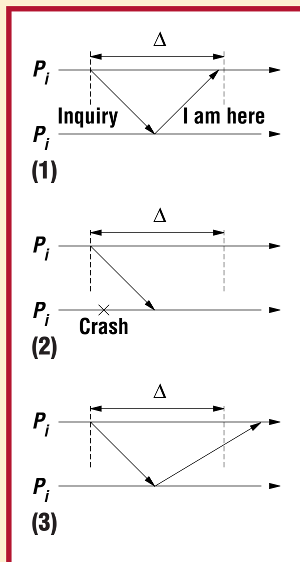


Figure B. Three scenarios of attempted communication in a distributed system: (1) communication is timely; (2) process p_j has crashed; (3) link (p_i, p_j) is slow.

lay + receiving-process scheduling delay + message-processing time." Such a value is a constant, known by all the system's sites.

Most real distributed systems are asynchronous in the sense that δ cannot be established.[5] So, asynchronous distributed systems are characterized by the absence of such an a priori known bound; message transfer and process-scheduling delays cannot be predicted and are considered arbitrary. Asynchronous distributed systems are sometimes also called *time-free* systems.

Crash failures

The underlying communication network is assumed to be reliable; that is, it doesn't lose, generate, or garble messages. The famous "Generals Paradox" shows that distributed systems with unreliable communication do not admit solutions to the NBAC problem.[2]

A site or transaction participant can fail by crashing. Its state is *correct* until it crashes. A participant that does not crash during the transaction execution is also said to be correct; otherwise, it is *faulty*. Because we are interested only in commitment and not in recovery, we assume that a faulty participant does not recover. Moreover, whatever the number of faulty participants and the network configuration, we assume that each pair of correct participants can always communicate. In synchronous systems, crashes can be easily and safely detected by using time-out mechanisms. This is not the case in asynchronous distributed systems.

The consensus problem

Consensus[6] is a fundamental problem of distributed systems (see the sidebar "Why Consensus Is Important in Distributed Systems"). Consider a set of processes that can fail by crashing. Each process has an input value that it proposes to the other processes. The consensus problem consists of designing a protocol where all correct processes unanimously and irrevocably decide on a common output value that is one of the input values.

The consensus problem comprises four properties:

- *Termination*. Every correct process eventually decides some value.
- *Integrity*. A process decides at most once.
- *Agreement*. No two correct processes decide differently.
- *Validity*. If a process decides a value, some process proposed that value.

The *uniform consensus problem* is defined by the previous four properties plus the *uniform agreement* property: no two processes decide differently.

When the only possible failures are process crashes, the consensus and uniform consensus problems have relatively simple solutions in synchronous distributed systems. Unfortunately, this is not the case in asynchronous distributed systems, where the most famous result is a negative one.

Authorized licensed use limited to: University of Minnesota. Downloaded on January 28, 2010 at 14:57 from IEEE Xplore. Restrictions apply.

Why Consensus Is Important in Distributed Systems

In asynchronous distributed systems prone to process crash failures, Tushar Chandra and Sam Toueg have shown that the consensus problem and the atomic broadcast problem are equivalent.[1] (The atomic broadcast problem specifies that all processes be delivered the same set of messages in the same order. This set includes only messages broadcast by processes but must include all messages broadcast by correct process.) Therefore, you can use any solution for one of these problems to solve the other. Suppose we have a solution to the atomic broadcast problem. To solve the consensus problem, each process "atomically broadcasts" its value and the first delivered value is considered the decision value. (See Chandra and Toueg's article[1] for information on a transformation from consensus to atomic broadcast). So, all theoretical results associated with the consensus problem are also applicable to the atomic broadcast problem. Among others, it is impossible to design an atomic broadcast protocol in an asynchronous distributed system prone to process crash failures without considering additional assumptions limiting the system's asynchronous behavior.

Also, in asynchronous distributed systems with limited behavior that make the consensus problem solvable, consensus acts as a building block on the top of which you can design solutions to other agreement or coordination problems (such as nonblocking atomic commitment, leader election, and group membership).

Reference

1. T.D. Chandra and S. Toueg, "Unreliable Failure Detectors for Reliable Distributed Systems," *J. ACM*, vol. 43, no. 2, Mar. 1996, pp. 245–267.

The *FLP* result states that it is impossible to design a deterministic protocol solving the consensus problem in an asynchronous system even with only a single process crash failure[6] (see the sidebar, "The Impossibility of Consensus in Asynchronous Distributed Systems"). Intuitively, this is due to the impossibility of safely distinguishing a very slow process from a crashed process in an asynchronous context. This impossibility result has motivated researchers to discover a set of minimal properties that, when satisfied by an asynchronous distributed system, make the consensus problem solvable.

Failure detection in asynchronous systems

Because message transfer and process-scheduling delays are arbitrary and cannot

The Impossibility of Consensus in Asynchronous Distributed Systems

Let's design a consensus protocol based on the rotating coordinator paradigm. Processes proceed by asynchronous rounds; in each round, a predetermined process acts as the coordinator. Let p_c be the coordinator in round r, where $c = (r \bmod n) + 1$. Here's the protocol's principle:

1. During round r (initially, $r = 1$), p_c tries to impose its value as the decision value.
2. The protocol terminates as soon as a decision has been obtained.

Let's examine two scenarios related to round r:

1. Process p_c has crashed. A noncrashed process p_i cannot distinguish this crash from the situation in which p_c or its communication channels are very slow. So, if p_i waits for a decision value from p_c, it will wait forever. This violates the consensus problem's termination property.
2. Process p_c has not crashed, but its communication channel to p_i is fast while its communication channels to p_j and $p_{c'}$ are very slow. Process p_i receives the value v_c of p_c and decides accordingly. Processes p_j and $p_{c'}$, after waiting for p_c for a long period, suspect that p_c has crashed and start round $r + 1$. Let $c' = ((r + 1) \bmod n) + 1$. During round $r + 1$, p_j and $p_{c'}$ decide on $v_{c'}$. If $v_c \neq v_{c'}$, the agreement property is violated.

This discussion shows that the protocol just sketched does not work. What is more surprising is that it is impossible to design a deterministic consensus protocol in an asynchronous distributed system even with a single process crash failure. This impossibility result, established by Michael Fischer, Nancy Lynch, and Michael Paterson,[1] has motivated researchers to find a set of properties that, when satisfied by an asynchronous distributed system, make the consensus problem solvable. Minimal synchronism,[2] partial synchrony,[3] and unreliable failure detectors[4] constitute answers to such a challenge. Researchers have also investigated randomized protocols to get nondeterministic solutions.[5]

References

1. M. Fischer, N. Lynch, and M. Paterson, "Impossibility of Distributed Consensus with One Faulty Process," *J. ACM*, vol. 32, no. 2, Apr. 1985, pp. 374–382.
2. D. Dolev, C. Dwork, and L. Stockmeyer, "On the Minimal Synchronism Needed for Distributed Consensus," *J. ACM*, vol. 34, no. 1, Jan. 1987, pp. 77–97.
3. C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the Presence of Partial Synchrony," *J. ACM*, vol. 35, no. 2, Apr. 1988, pp. 288–323.
4. T.D. Chandra and S. Toueg, "Unreliable Failure Detectors for Reliable Distributed Systems," *J. ACM*, vol. 43, no. 2, Mar. 1996, pp. 245–267.
5. M. Rabin, "Randomized Byzantine Generals," *Proc. 24th Symp. Foundations of Computer Science*, IEEE Press, Piscataway, N.J., 1983, pp. 403–409.

Authorized licensed use limited to: University of Minnesota. Downloaded on January 28, 2010 at 14:57 from IEEE Xplore. Restrictions apply.

Implementations of failure detectors can have only "approximate" accuracy in purely asynchronous distributed systems.

be bounded in asynchronous systems, it's impossible to differentiate a transaction participant that is very slow (owing to a very long scheduling delay) or whose messages are very slow (owing to long transfer delays) from a participant that has crashed. This simple observation shows that detecting failures with *completeness* and *accuracy* in asynchronous distributed systems is impossible. Completeness means a faulty participant will eventually be detected as faulty, while accuracy means a correct participant will not be considered faulty.

We can, however, equip every site with a failure detector that gives the site hints on other sites it suspects to be faulty. Such failure detectors function by suspecting participants that do not answer in a timely fashion. These detectors are inherently unreliable because they can erroneously suspect a correct participant or not suspect a faulty one. In this context, Tushar Chandra and Sam Toueg have refined the completeness and accuracy properties of failure detection.[7] They have shown that you can solve the consensus problem in executions where unreliable failure detectors satisfy some of these properties:

- *Strong completeness*. Eventually every faulty participant is permanently suspected by *every* correct participant.
- *Weak completeness*. Eventually every faulty participant is permanently suspected by *some* correct participant.
- *Weak accuracy*. There is a correct participant that is never suspected.
- *Eventual weak accuracy*. Eventually there is a correct participant that is never suspected.

Based on these properties, Chandra and Toueg have defined several classes of failure detectors. For example, the class of "eventual weak failure detectors" includes all failure detectors that satisfy weak completeness and eventual weak accuracy. We can use time-outs to meet completeness of failure detections because a faulty participant does not answer. However, as we indicated previously, accuracy can only be approximate because, even if most messages are received within some predictable time, an answer not yet received does not mean that the sender has crashed. So, implementations of failure detectors can only have "approximate" accuracy in purely asynchronous distributed systems. Despite such approximate implementations, unreliable failure detectors, satisfying only weak completeness and eventual weak accuracy, allow us to solve the consensus problem.[7]

Nonblocking atomic commitment

As we mentioned earlier, the NBAC problem consists of ensuring that all correct participants of a transaction take the same decision—namely, to commit or abort the transaction. If the decision is Commit, all participants make their updates permanent; if the decision is Abort, no change is made to the data (the transaction has no effect). The NBAC protocol's Commit/Abort outcome depends on the votes of participants and on failures. More precisely, the solution to the NBAC problem has these properties:

- *Termination*. Every correct participant eventually decides.
- *Integrity*. A participant decides at most once.
- *Uniform agreement*. No two participants decide differently.
- *Validity*. A decision value is Commit or Abort.
- *Justification*. If a participant decides Commit, all participants have voted Yes.

The obligation property

The previous properties could be satisfied by a trivial protocol that would always output Abort. So, we add the *obligation* property, which aims to eliminate "unexpected" solutions in which the decision would be independent of votes and of failure scenarios. The obligation property stipulates that if all participants voted Yes and everything went well, the decision must be Commit. "Everything went well" is of course related to failures. Because failures can be safely detected in synchronous systems, the obligation property for these systems is as follows:

> *S-Obligation*: If all participants vote Yes and there is no failure, the outcome decision is Commit.

Because failures can only be suspected, possibly erroneously, in asynchronous systems, we must weaken this property for the

Authorized licensed use limited to: University of Minnesota. Downloaded on January 28, 2010 at 14:57 from IEEE Xplore. Restrictions apply.

```
procedure NBAC (vote, participants)
begin
(1.1)  multicast (vote, participants);
(2.1)  wait ( (delivery of a vote No from a participant)
              or (∃ q ∈ participants: exception(q) has been notified to p)
(2.3)         or (from each q ∈ participants: delivery of a vote Yes)
(2.4)         );
(3.1)  case
(3.2)         a vote No has been delivered        → outcome := propose (Abort)
(3.3)         an exception has been notified      → outcome := propose (Abort)
(3.4)         all votes are Yes                   → outcome := propose (Commit)
(3.5)  end case
end
```

Figure 1. A generic nonblocking atomic commitment (NBAC) protocol.

problem to be solvable.[8] So, for these systems we use this obligation property:

> *AS-Obligation*: If all participants vote Yes and there is no failure suspicion, the outcome decision is Commit.

A generic NBAC protocol

Next, we describe a generic protocol[9] for the NBAC problem. In this protocol, the control is distributed: each participant sends its vote to all participants and no main coordinator exists.[4] Compared to a central coordinator-based protocol, it requires more messages but fewer communication phases, thus reducing latency. The protocol is described by the procedure `nbac (vote, participants)`, which each correct participant executes (see Figure 1). The procedure uses three generic statements (`multicast`, `exception`, and `propose`) whose instantiations are specific to synchronous or to asynchronous distributed systems. Each participant has a variable outcome that will locally indicate the final decision (Commit or Abort) at the end of the protocol execution.

A participant *p* first sends its vote to all participants (including itself) by using the `multicast` statement (Figure 1, line 1.1). Then *p* waits until it has been either

- delivered a No vote from a participant (line 2.1),
- notified of an `exception` concerning a participant *q* (line 2.2), or
- delivered a Yes vote from each participant (line 2.3).

At line 2.2, the notification `exception(q)` concerns *q*'s failure and will be instantiated appropriately in each type of system. Finally, according to the votes and the exception notifications that *p* has received, it executes the statement `outcome := propose(x)` (lines 3.1 to 3.5) with Commit as the value of *x* if it has received a Yes from all participants, and with Abort in all other cases.

We'll discuss `propose` in the next section.

Instantiations

Now we'll look at instantiations of the generic protocol for synchronous and asynchronous distributed systems (see Table 1).[9]

Synchronous systems

For these systems, the instantiations are `Rel_Multicast(v,P)`, `timer expiration`, and `x`.

Multicast(v,P). The `Rel_Multicast(v,P)` primitive allows a process to reliably send a message *m* to all processes of *P*. It has these properties:[3,10]

- *Termination*. If a correct process multicasts a message *m* to *P*, some correct process of *P* delivers *m* (or all processes of *P* are faulty).
- *Validity*. If a process *p* delivers a message *m*, then *m* has been multicast to a set *P* and *p* belongs to *P* (there is no spurious message).

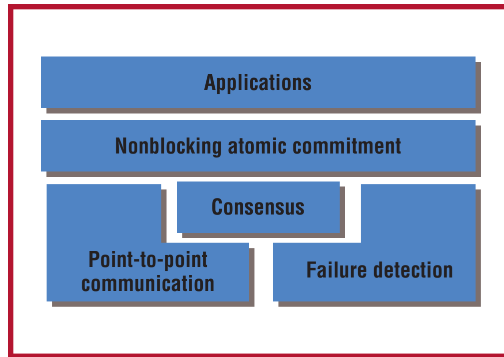
Table 1

Instantiations of the Generic Protocol for Distributed Systems

Generic statement	Synchronous instantiation	Asynchronous instantiation
`Multicast(v,P)`	`Rel_Multicast(v,P)`	`Multisend(v,P)`
`exception`	`timer expiration`	`failure suspicion`
`propose(x)`	`x`	`Unif_Cons(x)`

Authorized licensed use limited to: University of Minnesota. Downloaded on January 28, 2010 at 14:57 from IEEE Xplore. Restrictions apply.

Figure 2. A protocol stack.



- *Integrity*. A process p delivers a message m at most once (no duplication).
- *Uniform agreement*. If any (correct or not) process belonging to P delivers a message m, all correct processes of P deliver m.

The previous definition is independent of the system's synchrony. In synchronous distributed systems, the definition of `Rel_Multicast(v,P)` includes the additional property[3] of *timeliness*: there is a time constant Δ such that, if the multicast of m is initiated at real-time T, no process delivers m after $T + \Delta$.

Let f be the maximum number of processes that might crash and δ be the a priori known upper bound defined earlier in the section "Distributed systems and failures." We can show that $\Delta = (f + 1)\delta$. Özalp Babaoğlu and Sam Toueg describe several message- and time-efficient implementations of `Rel_Multicast(m,P)`.[3] So, in our instantiation, `Rel_Multicast` ensures that if a correct participant is delivered a vote sent at time T, all correct participants will deliver this vote by $T + \Delta$.

Exception. Because the crash of a participant q in a synchronous system can be safely detected, the `exception` associated with q is raised if q crashed before sending its vote. We implement this by using a single timer for all possible exceptions (there is one possible exception per participant): p sets a timer to $\delta + \Delta$ when it sends its vote (with the `Rel_Multicast` primitive). If the timer expires before a vote from each participant has been delivered, p can safely conclude that participants from which votes have not been delivered have crashed.

Propose(x). In this case, `propose` is simply the identity function—that is, `propose(x) = x`. So, `outcome := propose(x)` is trivially instantiated by `outcome := x`.

Asynchronous systems

For these systems, the instantiations are `Multisend(v,P)`, `failure suspicion`, and `Unif_Cons(x)`. These instantiations "reduce" the NBAC problem to the uniform consensus problem. This means that in asynchronous distributed systems, if we have a solution for the uniform consensus problem, we can use it to solve the NBAC problem[8] (see Figure 2).

Multicast(v,P). The primitive `Multisend(m,P)` is a syntactical notation that abstracts `for each p ∈ P do send(m) to p end do` where `send` is the usual point-to-point communication primitive. `Multisend(m,P)` is the simplest multicast primitive. It is not fault tolerant: if the sender crashes after having sent message m to a subset P' of P, message m will not be delivered to all processes belonging to P but not to P'. In our instantiation, if a participant crashes during the execution of `Multisend`, it will be suspected by any failure detector that satisfies the completeness property.

Exception. When the failure detector associated with participant p suspects (possibly erroneously) that a participant q has crashed, it raises the `exception` associated with q by setting a local Boolean flag `suspected(q)` to the value `true`. If failure detectors satisfy the completeness property, all participants that crashed before sending their vote will be suspected. (The protocol's termination is based on this observation.)

Propose(x). The function `propose` is instantiated by any subprotocol solving the uniform consensus problem. Let Unif_Cons be such a protocol; it is executed by all correct participants.[5]

The study of the NBAC problem teaches us two lessons. In the presence of process crash failures, the first lesson comes from the generic statement `propose`. In a synchronous system, a correct participant can locally take a `globally consistent` decision when a timer

Authorized licensed use limited to: University of Minnesota. Downloaded on January 28, 2010 at 14:57 from IEEE Xplore. Restrictions apply.

expires. In an asynchronous system, the participant must cooperate with others to take this decision.

The second lesson comes from failure detectors. They let us precisely characterize the set of executions in which we can solve the NBAC problem in purely asynchronous systems. Weak completeness and eventual weak accuracy delineate the precise frontier beyond which the consensus problem cannot be solved.[11]

To master the difficulty introduced by the detection of failures in distributed systems, it's necessary to understand the few important notions we've presented. These notions should be helpful to researchers and engineers to state precise assumptions under which a given problem can be solved. In fact, the behavior of reliable distributed applications running on asynchronous distributed systems should be predictable in spite of failures.

About the Authors

Michel Raynal is a professor of computer science at the University of Rennes, France. He is involved in several projects focusing on the design of large-scale, fault-tolerant distributed operating systems. His research interests are distributed algorithms, operating systems, parallelism, and fault tolerance. He received his Doctorat dEtat en Informatique from the University of Rennes. Contact him at IRISA Campus de Beaulieu, 35042 Rennes Cedex, France; raynal@irisa.fr.

Mukesh Singhal is a full professor of computer and information science at Ohio State University, Columbus. He is also the program director of the Operating Systems and Compilers program at the National Science Foundation. His research interests include operating systems, database systems, distributed systems, performance modeling, mobile computing, and computer security. He has coauthored *Advanced Concepts in Operating Systems* (McGraw-Hill, 1994) and *Readings in Distributed Computing Systems* (IEEE CS Press, 1993). He received his Bachelor of Engineering in electronics and communication engineering with high distinction from the University of Roorkee, Roorkee, India, and his PhD in computer science from the University of Maryland, College Park. He is a fellow of the IEEE. Contact him at the Dept. of Computer and Information Science, Ohio State Univ., Columbus, OH 43210; singal@cis.ohio-state.edu; www.cis.ohio-state.edu/~singhal.

References

1. P.A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, Reading, Mass., 1987.
2. J.N. Gray, "Notes on Database Operating Systems," *Operating Systems: An Advanced Course*, *Lecture Notes in Computer Science*, no. 60, Springer-Verlag, Heidelberg, Germany, 1978, pp. 393–481.
3. Ö. Babaoğlu and S. Toueg, "Non-Blocking Atomic Commitment," *Distributed Systems*, S. Mullender, ed., ACM Press, New York, 1993, pp. 147–166.
4. D. Skeen, "Non-Blocking Commit Protocols," *Proc. ACM SIGMOD Int'l Conf. Management of Data*, ACM Press, New York, 1981, pp. 133–142.
5. F. Cristian, "Understanding Fault-Tolerant Distributed Systems," *Comm. ACM*, vol. 34, no. 2, Feb. 1991, pp. 56–78.
6. M. Fischer, N. Lynch, and M. Paterson, "Impossibility of Distributed Consensus with One Faulty Process," *J. ACM*, vol. 32, no. 2, Apr. 1985, pp. 374–382.
7. T.D. Chandra and S. Toueg, "Unreliable Failure Detectors for Reliable Distributed Systems," *J. ACM*, vol. 43, no. 2, Mar. 1996, pp. 245–267.
8. R. Guerraoui, "Revisiting the Relationship between Non-Blocking Atomic Commitment and Consensus," *Proc. 9th WDAG*, *Lecture Notes in Computer Science*, no. 972, Springer-Verlag, Heidelberg, Germany, 1995, pp. 87–100.
9. M. Raynal, "Non-Blocking Atomic Commitment in Distributed Systems: A Tutorial Based on a Generic Protocol," to be published in *Int'l J. Computer Systems Science and Eng.*, vol. 15, no. 2, Mar. 2000, pp. 77–86.
10. V. Hadzilacos and S. Toueg, "Reliable Broadcast and Related Problems," *Distributed Systems*, 2nd ed., S. Mullender, ed., ACM Press, New York, 1993, pp. 97–145.
11. T.D. Chandra, V. Hadzilacos, and S. Toueg, "The Weakest Failure Detector for Solving Consensus," *J. ACM*, vol. 43, no. 4, July 1996, pp. 685–822.

Authorized licensed use limited to: University of Minnesota. Downloaded on January 28, 2010 at 14:57 from IEEE Xplore. Restrictions apply.