Consensus

Reaching agreement

Paul Krzyzanowski *https://www.cs.rutgers.edu/~pxk/417/notes/content/consensus.html* October 2015

Introduction

Consensus is the task of getting all processes in a group to agree on some specific value based on the votes of each processes. All processes must agree upon the same value and it must be a value that was submitted by at least one of the processes (i.e., the consensus algorithm cannot just invent a value).

This is a simple-sounding problem but finds a surprisingly large amount of use in distributed systems. Any algorithm that relies on multiple processes maintaining common state relies on solving the consensus problem. Some examples of places where consensus is useful are:

- synchronizing replicated state machines and making sure all replicas have the same (consistent) view of system state.
- electing a leader (e.g., for mutual exclusion)
- distributed, fault-tolerant logging with globally consistent sequencing
- managing group membership
- deciding to commit or abort for distributed transactions

Consensus among processes is easy to achieve in a perfect world. For example, when we examined distributed mutual exclusion algorithms earlier, we visited a form of consensus where everybody reaches the same decision on who can access a resource. The simplest implementation was to assign a system-wide coordinator who is in charge of determining the outcome. This is easy because we assume that all processes are functioning and are able to communicate with each other. Faulty processes, computers, and networks make consensus challenging.

Synchronous vs. Asynchronous Systems

A synchronous system is one where one process sends a message to the other within a bounded time. This means there is a time limit for the transit of a message as well as for the operating system to schedule the sending process to dipatch the message and for the other process to receive it. A direct serial port connection between two computers or the public switched telephone network are examples of a synchronous communication link.

In an asynchronous system, there are no prescribed time limits for message arrival. We might have expectations for delivery within a certain time but the system will not guarantee that these expectations will be met. The Internet is an example of an asynchronous system.

In building a distributed system, we assume that processes are concurrent, asynchronous, and subject to failure.

Dealing with failure

We cannot provably achieve consensus with completely asynchronous faulty processes. This means making no assumption on the speeds of the processes or network communication. The core problem is that there is no way to check whether a process has failed or whether the process is alive but the communication to the process is intolerably slow. This impossibility is proved by Fischer, Lynch, Patterson (FLP[85]). Also, in the presence of unreliable communication, consensus is impossible to achieve since we may never be able to communicate with a process.

We will examine two fault tolerance scenarios that illustrate some basic constraints that are imposed on us. The two army problem is particularly relevant.

Two Army Problem

Let us examine the case of good processors but faulty communication lines. This is known as the two army problem and can be summarized as follows:

Two divisions of an army, *A* and *B*, coordinate an attack on enemy army, *C*. *A* and *B* are physically separated and use a messenger to communicate. *A* sends a messenger to *B* with a message of "*let's attack at dawn*". *B* receives the message and agrees, sending back the messenger with an "*OK*" message. The messenger arrives at *A*, but *A* realizes that *B* did not know whether the messenger made it back safely. If *B* is not convinced that *A* received the acknowledgement, then it will not be confident that the attack should take place since the army will not win on its own. *A* may choose to send the messenger back to *B* with a message of "*A received the OK*" but *A* will then be unsure as to whether *B* received this message. The two army problem demonstrates that even with non-faulty processors, provable agreement between two processes is *not* possible with unreliable communication channels.

In the real world, we will need to place upper bounds on communication and computing speeds and consider a process to be faulty if it does not respond within that bounded time. Not only that, but we need to account for the possibility that the process is not faulty and may continue trying to participate in the computation.

Fail-stop, also known as fail-silent, is the condition when a failed process does not communicate. A byzantine fault is where the faulty process continues to communicate but may produce faulty information. We can create a consensus algorithm that is resilient to fail-stop. If there are *n* processes, of which *t* may be faulty, then a process can never expect to receive more than *(n-t)* acknowledgements. The consensus problem now is to make sure that the same decision is made by all processes, even if each process receives up to *(n-t)* answers from a different set of processes (perhaps due to partial network segmentation or routing problems).

A fail-stop resilient algorithm can be demonstrated as follows. It is an iterative algorithm. Each phase consists of:

1. A process broadcasts its preferred value and the number of processes that it has seen that also have that preferred value (this count is called the cardinality and is 1 initially).
2. The process receives *(n-t)* answers, each containing a preferred value and cardinality.
3. The process may then change its preferred value according to which value was preferred most by other processes. It updates the corresponding cardinality to the number of responses it has received with that value plus itself.

Continue this process until a process receives *t* messages of a single value with cardinality at least *n/2*. This means that at least half of the systems have agreed on the same value. At this point, run two more phases, broadcasting this value.

As the number of phases goes to infinity, the probability that consensus not reached approaches 0.

To make it easier to develop algorithms in the real world, we can relax the definition of *asynchronous* and allow some synchrony. Several types of asynchrony may exist in a system:

1. Process asynchrony: a process may go to sleep or be suspended for an arbitrary amount of time.
2. Communication asynchrony: there is no upper bound on the time a message may take to reach its destination.
3. Message order asynchrony: messages may be delivered in a different order than sent.

It has been shown [Dolev, D., Dwork, C., and Stockmeyer] that it is not sufficient to make processes synchronous but that any of the following cases is sufficient to make a consensus protocol possible:

1. Process and communication synchrony: place an upper bound on process sleep time and message transmission).
2. Process and message order synchrony: place an upper bound on process sleep time and message order).
3. Message order synchrony and broadcast capability.
4. Communication synchrony, broadcast capability, and send/receive atomicity. *Send/receive atomicity* means that a processor can carry through the operations of receiving a message, performing computation, and sending messages to other processes.

Byzantine failures in synchronous systems

> Solutions to the Byzantine Generals problem are *not* obvious, intuitive, or simple. They are not presented in these notes. You can read Lamport's paper on the problem here. You can also check out the brief summary and various solutions – which go beyond the Lamport paper – here.

We looked at the case of unreliable communication lines with reliable or fail-stop communication. The other case to consider is that of reliable communication lines but faulty (not fail-stop) processors. Byzantine failures are failed processors that, instead of staying quiet (fail-stop), instead communicate with erroneous data.

Byzantine Generals Problem

Consensus with reliable communication lines and byzantine failures is illustrated by the Byzantine Generals Problem. In this problem, there are *n* army generals who head different divisions. Communication is reliable (radio or telephone) but *m* of the generals are traitors (faulty) and are trying to prevent others from reaching agreement by feeding them incorrect information. The question is: can the loyal generals still reach agreement? Specifically, each general knows the size of his division. At the end of the algorithm can each general know the troop strength of every other loyal division?

Lamport demonstrated a solution that works for certain cases. His answer to this problem is that any solution to the problem of overcoming *m* traitors requires a minimum of 3*m*+1 participants (2*m*+1 loyal generals). This means that more than 2/3 of the generals must be loyal. Moreover, it was demonstrated that no protocol can overcome *m* faults with fewer than *m*+1 rounds of message exchanges and $O(mn^2)$ messages. If $n < 3m + 1$ then the problem has no solution.

Clearly, this is a rather costly solution. While the Byzantine model may be applicable to certain types of special-purpose hardware, it will rarely be useful in general purpose distributed computing environments.

There is a variation on the Byzantine Generals Problem that uses signed messages. What this means is that messages from loyal generals cannot be forged or modified. In this case, there are algorithms that can achieve consensus for values of $n \geq m + 2$, where

n = total number of processors

m = total number of faulty processors

Replicated state machines

An important motivation for building distributed systems is to achieve high scalability and high availability. High availability can be achieved via redundancy: replicated functioning components will take the place of those that ceased to function. To achieve redundancy with multiple active components, we want all working replicas to do the same thing: produce the same outputs given the same inputs.

A state machine approach to systems design models each replica (each component of the system) as a deterministic state machine. For some given input to a specific state of the system, a deterministic output and transition to a new state will be produced. We refer to each replica (component) as a *process*. For correct execution and high availability, it is important that each process sees the same inputs. To do this, we rely on a consensus algorithm. This ensures that multiple processes will do the same thing since they will each be provided with the same set of inputs.

An example of an input may be a request from a client to read data from a specific location from a file or write data to a specific location of a file. We want the replicated files to contain the exact same data and yield the same results. To achieve this, we need agreement among all processes on what the client requests are and the requests must be totally ordered: each server must see file read/write requests in the exact same order as everyone else. The total ordering part is most easily achieved by electing one process to serve sequence numbers (although there are other more complex but more distributed implementations.

Paxos

Paxos is a popular and widely-used fault-tolerant distributed consensus algorithm. It allows a globally consistant (total) order to be assigned to client messages (actions).

Much of what is summarized here is from Lamport's *Paxos Made Simple* but I tried to simplify it substantially. Please refer to that paper for more detail and definitive explanations.

The goal of a distributed consensus algorithm is to allow a set of computers to all agree on a single value that one of the nodes in the system proposed (as opposed to making up a random value). The challenge in doing this in a distributed system is that messages can be lost or machines cn fail. Paxos guarantees that a set of machines will chose a single proposed value as long as a majority of systems that participate in the algorithm are available.

The setting for the algorithm is that of a collection of processes that can propose values. The algorithm has to ensure that a single one of those proposed values is chosen and all processes should learn that value.

There are three classes of agents:

1. Proposers
2. Acceptors
3. Learners

A machine can take on any or all of these roles. Typically, learners will be integrated with acceptors, for example. Proposers put forth proposed values. Acceptors drive the algorithm's goal to reach agreement on a single value and let the learners are informed of the outcome. Acceptors either reject a proposal or agree to it and make promises on what proposals they will accept in the future. This ensures that only the latest set of propsals will be accepted. A process can act as more than one agent in an implementation. Indeed, many implementations have collections of processes where each process takes on all three roles.

Agents communicate with each other asynchronously. They may also fail to communicate and may restart. Messages can take arbitrarily long to deliver. They may can be duplicated or lost but are not corrupted. A corrupted message should be detectable as such and can be counted as a lost one (this is what UDP does, for example).

The absolutely simplest implementation contains a single acceptor. A proposer sends a proposal value to the acceptor. The acceptor processes one request at a time, chooses the first proposed value that it receives, and lets everyone (learners) know. Other proposers must agree to that value.

This works as long as the acceptor does not fail. Unfortunately, acceptors are subject to failure. To guard against the failure of an acceptor, we turn to replication and use multiple acceptor processes. A proposer now sends a *proposal* containing a value to a set of acceptors. The value is chosen when a majority of the acceptors *accept* that proposal (agree to it).

Different proposers, however, could independently initiate proposals at approximately the same time and those proposals could contain different values. They each will communicate with a different subset of acceptors. Now different acceptors will each have different values but none will have a majority. We need to allow an acceptor to be able to accept more than one proposal. We will keep track of proposals by assigning a unique proposal number to each proposal. Each proposal will contain a proposal number and a value. The value is the thing on which we need to agree; for example setting a *name=value* field in a replicated database. Each proposal must have a unique proposal number. Our goal is to agree on one of those proposed values from the pool of proposals sent to different subsets of acceptors.

A value is chosen when a single proposal with that value has been accepted by a majority of the acceptors. That means it has been *chosen*. Multiple proposals can be chosen but all of them bust have the same value: if a proposal with a value *v* is chosen, then every higher-numbered proposal that is chosen must also have value *v*.

If a proposal with proposal number *n* and value *v* is issued, then there is a set *S* consisting of a majority of acceptors such that either:

a. no acceptor in S has accepted any proposal numbered less than n, or
b. v is the value of the highest-numbered proposal among all proposals numbered < n accepted by the acceptors in S.

A proposer that wants to issue a proposal numbered *n* must learn the highest numbered proposal with number less than *n*, if any, that has been or will be accepted by each acceptor in a majority of acceptors. To do this, the proposer gets a *promise* from an acceptor that there will be no future acceptance of proposals numbered less than *n*.

The Paxos algorithm

With Paxos, a client sends a proposal (e.g., a *name=value* setting) to a *proposer*, which is then responsible for running the algorithm. The Paxos algorithm operates in two phases:

Phase 1: PREPARE: send a proposal request

A. Proposer: prepare

- A proposer chooses a proposal number *N* and sends a *prepare* request to a majority of acceptors. The number *N* is stored in the proposer's stable storage so that the proposer can ensure that a higher number is used for the next proposal (even if the proposer process restarts). To ensure uniqueness among all proposers, the proposal number can be of the form *sequence_number.process_id*, where *sequence_number* is a monotonically-increasing local number and *process_id* is a unique identifier for the process, such as the machine's IP or Ethernet MAC address concatenated with its process ID.

- The proposer sends a prepare(N) message to the majority of acceptors.

B. Acceptor: promise – receive a *prepare(N)* message

- The acceptor promises not to accept any *prepare* messages with smaller request numbers. If an acceptor has already received a proposal greater than *N*, it will reject this *prepare(N)* request. To do this, it keeps track of the highest proposal number that it has seen.
  - `if (N > max_received_proposal)`
  - `  max_received_proposal = N`
  - `else`
  - `  reject()`
- It is possible that the acceptor may have already accepted a proposal (e.g., one that came concurrently from another proposer). In this case, it will convey that information to the proposer. To be able to do this, the acceptor keeps track of the highest previously accepted proposal number and its value.
  - `if (have_accepted_proposal)`
  - `  promise(accepted_number, accepted_value)`
  - `else`
  - `  promise(N)`

C. Proposer: receive *promise* messages from a majority of acceptors

- If a proposer receives *promise* message from a majority of acceptors, it can now choose a value. If *any* of the acceptors returned an accepted proposal, the proposer chooses the one associated with the highest proposal number. The proposer *must* use this value instead of the one it originally proposed. Note that the proposer changes its value but does not change its proposal number in this case. If no acceptor returned an accepted value, then the proposer is free to use the value it originally proposed along with its proposal number.

Phase 2: ACCEPT**: send a proposal (and then propagate it to learners after acceptance)**

Proposer:

- A proposer can now issue its proposal. Note that the value is either the proposer's initial value or a value it received from an acceptor. It will send a message to all acceptors (reaching a majority): accept(N, value)

Acceptor:

- When an acceptor receives an *accept(N, value)* message, it checks to see if it is the highest sequence number that it has seen. If so, then it accepts the proposal and stores information about the request so it can return it to other proposers, if necessary. It also sends the proposal value to each *learner* node. Note that, in some implementations, the learner may implemented at the proposer. In this case, that proposed value is returned to the acceptor.
  -
  - `if (N ≥ max_received_proposal) {`
  - `  accepted_value = value`
  - `  accepted_number = N`
  - `  max_received_proposal = N`
  - `  have_accepted_proposal = true`
  - `}`
  - `return max_received_proposal (or send to learner)`

In all cases, the acceptor returns the maxium received proposal number.

Acceptor:

- The learner (or proposer, if it implements the learner's funciton) must receive responses from a majority of acceptors. If a proposer receives response contains a proposal number that is greater than the proposal number it submitted, then it knows that that request has been rejected.

The acceptor receives two types of requests from proposers: *prepare* and *accept* requests. Any request can be ignored. An acceptor only needs to remember the highest-numbered proposal that it has ever accepted and the number of the highest-numbered *prepare* request to which it has responded. The acceptor must store these values in stable storage so they can be preserved in case the acceptor fails and has to restart.

A proposer can make multiple proposals as long as it follows the algorithm for each one.

Consensus

Now that the acceptors have a proposed value, we need a way to learn that a proposal has been accepted by a majority of acceptors. The *learner* is responsible for getting this information, although its role is often integrated into the proposer process. Each acceptor, upon accepting a proposal, forwards it to all the learners. The problem with doing this is the potentially large number of duplicate messages: *(number of acceptors) * (number of learners)*. If desired, this could be optimized. One or more *"distinguished learners"* could be elected. Acceptors will communicate to them and they, in turn, will inform the other learners.

Ensuring progress

One problem with the algorithm is that its possible for two proposers to keep issuing sequences of proposals with increasing numbers, none of which get chosen. An *accept* message from one proposer may be ignored by an acceptor because a higher numbered *prepare* message has been processed from the other proposer. To ensure that the algorithm will make progress, a *"distinguished proposer"* is selected as the only one to try issuing proposals.

In operation, clients send commands to the leader, an elected *"distinguished proposer"*. This proposer sequences the commands (assigns a value) and runs the Paxos algorithm to ensure that an agreed-upon sequence number gets chosen. Since there might be conflicts due to failures or another server thinking it is the leader, using Paxos ensures that only one command (proposal) gets assigned that value.

Leasing versus Locking

Processes often rely on locks to ensure exclusive access to a resource. The difficulty with locks is that they are not fault-tolerant. If a process holding a lock dies or forgets to release the lock, the lock exists unless additional software is in place to detect these actions and break the lock. For this reason, it is more safer to add an expiration time to a lock. This turns a *lock* into a *lease*.

We saw an example of this approach with the two-phase and three-phase commit protocols. A two-phase commit protocol uses locking while the three-phase commit uses leasing; if a lease expires, the transaction is aborted. We also saw this approach with maintaining references to remote objects. If the lease expires, the server considers the object unreferenced and suitable for deletion. The client is responsible for renewing the lease periodically as long as it needs the object.

The downside with a leasing approach is that the resource is unavailable to others until the lease expires. Now we have a trade-off: have long leases with a possibly long wait after a failure or have short leases that need to be renewed frequently.

Hierarchical leases versus consensus

In a fault tolerant system with replicated components, leases for resources should be granted by running a consensus algorithm. Looking at Paxos, it is clear that, while there is not a huge amount of message passing taking place, there are number of players involved and hence there is a certain efficiency cost in using the algorithm. A compromise approach is to use the consensus algorithm as an election algorithm to elect a

coordinator. This coordinator is granted a lease on a large set of resources or the state of the system. In turn, the coordinator is now responsible for handing out leases for all or a subset of the system state. When the coordinator's main lease expires, a consensus algorithm has to be run again to grant a new lease and possibly elect a new coordinator but it does not have to be run for every client's lease request; that is simply handled by the coordinator.

References

Leslie Lamport, *Paxos Made Simple*, November 2001.

One of the clearest papers out there detailing the Paxos algorithm

Angus MacDonald, *Paxos By Example*, *June 27, 2012.*

Short and clear walkthough of an example of using Paxos to achieve consensus.

Lampson, Butler. How to Build a Highly Available System Using Consensus, Microsoft Research

An updated version of Distributed Algorithms, ed. Babaoglu and Marzullo, Lecture Notes in Computer Science 1151, Springer, 1996, pp 1-17.

A great coverage of leases, the Paxos algorithm, and the need for consensus in achieving highly available computing using replicated state machines.

Henry Robinson, Consensus Protocols: Paxos, Paper Trail blog, February 2009.

Iair Amir, Jonathan Kirsch, Paxos for System Builders: An Overview, Johns Hopkins University.

Written from a system-builder's perspective and covers some of the details of implementation. The paper is a really brief (5 page) overview.

Iair Amir, Jonathan Kirsch, Paxos for System Builders, Johns Hopkins University, Technical Report CNDS-2008-2, March 2008.

This is the 35-page full version of the above paper.

Michael J. Fischer, Nancy A. Lynch, Michael S. Paterson, Impossibility of Distributed Consensus with One Faulty Process, Journal of the Association for Computing Machinery, Volume 32, No. 2, April 1985, pp. 374-382.

This is the seminal paper (known as FLP85) that proves that one cannot achieve consensus with completely asynchronous faulty processes.

Bracha, G. and Toueg, S.** Asynchronous Consensus and Broadcast Protocols, **Journal of the ACM 32, 4 (October 1985), 824–840.

Describes a fail-stop consensus algorithm.

Dolev, D., Dwork, C., and Stockmeyer, L.** On the Minimal Synchronism Needed for Distributed Consensus, **J. ACM 34, 1 (January 1987), 77–97.

This is an updated version of the original that was published on October, 2011.