Distributed Algorithms

Benoît Garbinato
distributed object programming lab

Distributed systems

"As long as there were no ~~machines~~ networks, distributed programming was no problem at all; when we had a few weak ~~computers~~ networks, distributed programming became a mild problem and now that we have gigantic ~~computers~~ networks, distributed programming has become an equally gigantic problem. In this sense the electronic industry has not solved a single problem, it has only created them – it has created the problem of using its products."

Edgster Dijkstra, The Humbel Programmer.
Communication of the ACM, vol. 15, no. 10.
October 1972. Turing Award Lecture.

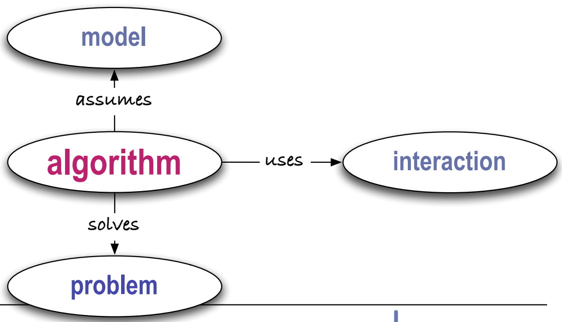
Our approach

- The practitioner needs the theoretical perspective to understand the implicit assumptions hidden in the technologies, and their consequences
- The theoretician needs the practical perspective to validate that theoretical models, problems & solutions work in accordance to existing technologies
- To achieve this, we approached distributed systems through trhee complementary views:
  - The model view
  - The interaction view
  - The algorithm view

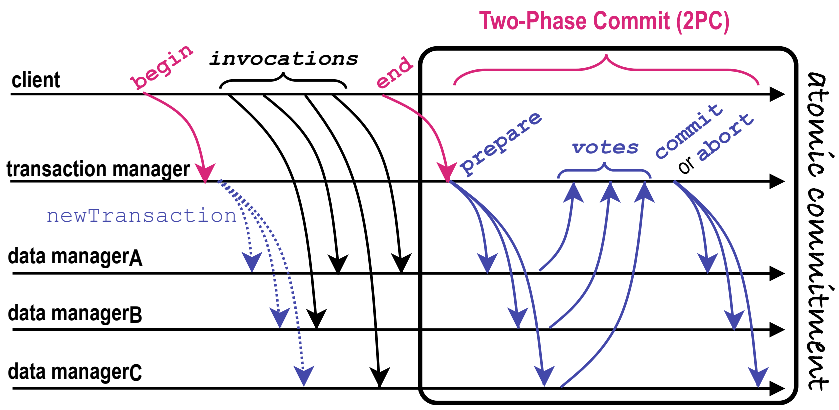
The big picture

When implementing a distributed program, you will always end up writing some algorithm. In doing so, you will have to answer the following questions:

- What problem am I trying to solve?
- What model do I assume?
- What interaction do I use?



Atomic commitment



Here are the actual two phases

Problem: atomic commitment
Interaction: reliable message passing
Model: synchronous crash-recovery
Algorithm: 2-phase commit protocol

A few observations

- Most atomic commitment protocols guarantee that safety will always hold, but not necessarily liveness
- Liveness is compromised when failures prevent the Termination property from holding; in such a case, we say that the protocol is blocking
- In the crash-recovery model, a blocking protocol cannot terminate until crashed processes have recovered
- Upon recovery, a failed process reads it log file from stable storage and acts according to its last operation
- In atomic commitment terms, this implies that the recovering process should be able to decide commit or abort from what it finds in its log file

Agreement problems

- The atomic commitment is an instance of a more general agreement problem, also known as the consensus problem
- There exists many variants of the consensus problem, which are not necessarily equivalent to each other

Problem specification

The atomic commitment problem corresponds to the following consensus variant, with the transaction manager and data managers being processes, value 1 corresponding to commit and value 0 corresponding to abort

<u>Agreement</u> *(safety property)*

No two processes decide on different values

<u>Validity</u> *(safety property)*

- If any process starts with 0, then 0 is the only possible decision
- If all processes start with 1 and there are no failures, then 1 is the only possible decision

<u>Termination</u> *(liveness property)*

<u>Weak</u>: if there are no failures, then all processes eventually decide
<u>Strong</u>: all non faulty processes eventually decide

Two-phase commit (2PC)

Premises:

- synchronous model, reliable channels
- crash-recovery failures of data managers D_i
- transaction manager T acts as coordinator but also votes

Phase 1:

- each D_i process sends its initial value to process T
- any process D_i whole initial value is 0 decides 0
- if process T times out waiting for some initial value, it decides 0; otherwise it decides for the minimum of all values

Phase 2:

- process T broadcasts its decision to all D_i processes
- any process that has not yet decided adopts this decision

What Termination property is ensured?

Upon recovery (2PC)

Premises:

- operations are logged onto stable storage before execution
- the logging of an operation and its execution are atomic

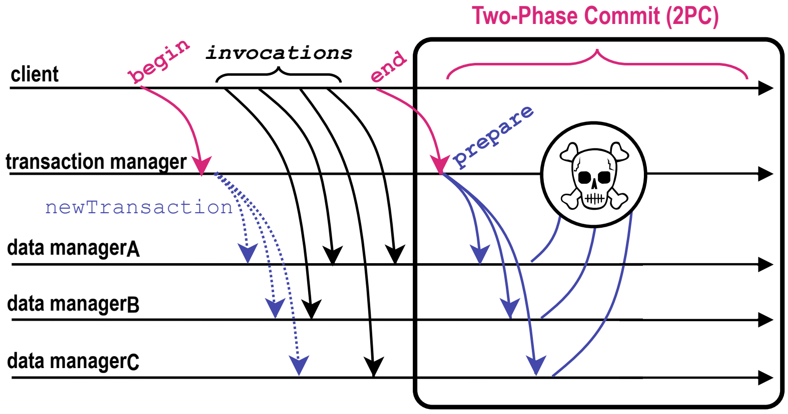
Recovery of a D_i process:

D_i reads its log file from stable storage

- if it voted 0 or if it crashed before sending its vote to T, it aborts
- otherwise, it asks T for the outcome of the transaction and acts accordingly

Limits of 2PC

<u>Questions</u>: what happens if the transaction manager crashes before sending the final commit or abort message?



If process T crashed...

<u>Case A</u>: some process has decided 0

⇒ it knows that T has either not decided or that it has decided 0

⇒ it can inform all other D_i process that it is safe to decide 0

<u>Case B</u>: all D_i processes have voted 1

⇒ no decision is possible (blocking) :

1. T might have decided 0, so deciding 1 violates Agreement
2. T might have decided 1, so deciding 0 violates Agreement

Three-phase commit (3PC)

Premises:
- synchronous model, reliable channels
- crash-recovery failures of any process
- transaction manager T acts as coordinator but also votes

Phase 1:
- each D_i process sends its initial value to process T
- any process D_i whole initial value is 0 decides 0
- if process T times out waiting for some initial value or receives 0 from some process, it decides 0; otherwise it goes to ready state

Phase 2:
- if process T decided 0, it broadcasts its decision to all D_i processes, so any process that has not yet decided adopts this decision
- if process T is ready state, it broadcasts a pre-commit message, so all processes go to ready state and send an ack message to T
- if process T crashes, the other processes time out and decide 0

Phase 3:
- if process T receives ack messages from all processes, it decides 1 and broadcast its decision, so all processes decide 1 as well
- if process T time out waiting for some ack message, it decides 0 and broadcast its decision, so all processes decide 0 as well
- if process T crashes, the other processes time out and decide 1

Upon recovery (3PC)

Premises: same as 2PC

Recovery of a D_i process:

D_i reads its log file from stable storage
- if it voted 0 or if it crashed before acknowledging the pre-commit message, it aborts
- otherwise, it asks T for the outcome of the transaction and acts accordingly

Recovery of T:

T reads its log file from stable storage:
- if it crashed before receiving all ack messages, it aborts
- otherwise, it commits

Limits of 3PC

If T fail in Phase 3, no other process is allowed to fail

Problematic scenario in Phase 3:

1. some D_i crashes before acknowledging pre-commit message
2. T decides 0 but crashes before broadcasting its decision
3. all other D_i time out waiting for the decision and decide 1

⇒ Agreement is violated!

Why not have all other D_i decide 0 then?

Further problems

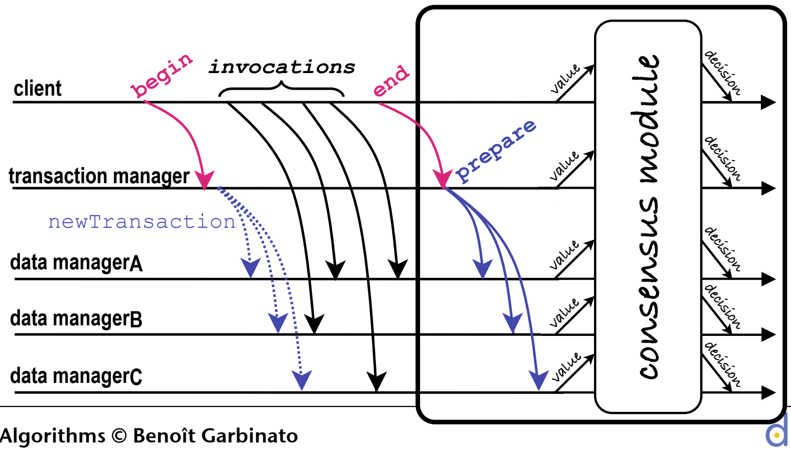
Unrealistic assumptions: synchronous processes and network, no network partitions
⇒ reliable failure detection

Drastic limitation on failures: see previous slide
⇒ T is a single point of failure/vulnerability

Hidden assumptions: logging an action & executing it must be atomic, deciding & broadcasting the decision must also be atomic

Back to consensus

- If we express the atomic commitment protocol in terms of some consensus module, we can benefit from all the algorithmic work done on the subject



Consensus & asynchrony

- Consensus cannot be solved in asynchronous systems; this is the famous Fisher-Lynch-Paterson (FLP) impossibility result
- For atomic commitment, the FLP result implies that we cannot answer the question "how long should we wait before aborting?"
  - if we do not wait long enough, safety is at stake
  - if we wait forever, liveness is at stake
- Real distributed systems are partially synchronous, i.e., they are mostly synchronous but they experience asynchronous periods every now and then. So, if we can solve a given problem during a synchronous period, that's all we need.

Failure detectors

- A failure detector is a module that provides each process with hints about possible crashes of other processes
- A failure detector encapsulates time assumptions and turns them into logical properties: completeness & accuracy. For example, the eventually strong failure detector ($\Diamond S$) ensures:

  Strong Completeness. Eventually, every process that crashes is permanently suspected by every correct process.
  Eventual Weak Accuracy. Eventually, there exists a correct process that is never suspected by any correct process

- The actual implementability of a given failure detector depends on the underlying timing assumption

Failure detectors & consensus

- The $\Diamond S$ failure detector was proven to be the weakest failure detector to solve consensus, provided that there are less than half incorrect processes
- The algorithm relies on the rotating coordinator paradigm, where a different process has the opportunity to become the next coordinator each time the current coordinator is suspected to have crashed
- The Strong Completeness of $\Diamond S$ ensures that no process will wait forever for the decision of a crashed coordinator
- The Eventual Weak Accuracy of $\Diamond S$ ensures that at least one of the coordinators will be able to decide

Fault-tolerant broadcasts

- The ability to broadcast messages with some dependable guarantees is a key issue when building fault-tolerant distributed systems
- Besides the reliable delivery of messages, there ordering is another aspect of this issue
- For example, if messages represents updates sent to the replicas of a database, reliable delivery and total ordering are necessary

Reliable broadcast (basis)

In the following, we assume that each message m includes (1) the identity of the sender, written sender(m) , and (2) a sequence number, denoted seq#(m). These two fields are what makes each message unique.

Validity

If a correct process broadcasts a message m, then it eventually delivers m

Agreement

Standard: If a correct process delivers a message m, then all correct processes eventually deliver m

Uniform: If a process delivers a message m, then all correct processes eventually deliver m

Integrity

For any message m, every correct process delivers m at most once, and only if m was previously broadcasted by sender(m)

Fifo broadcast

To obtain the specification of fifo broadcast, we simply add the following fifo order property to the aforementioned validity, agreement and integrity properties. That is, fifo broadcast ⇔ reliable broadcast + fifo order

Fifo order

If a process broadcasts a message m before it broadcasts a message m', then no correct process delivers m' unless it has previously delivered m

Atomic broadcast (total order)

To obtain the specification of atomic broadcast, we simply add the following total order property to the aforementioned validity, agreement and integrity properties. That is, atomic broadcast ⇔ reliable broadcast + total order

Total order

If correct processes p and q both deliver messages m and m', then p delivers m before m' if and only if q delivers m before m'

Question: does this imply Fifo Order ?

Causal broadcast

- Very often, perfectly synchronized clocks are not available, due to drifts, impreciseness, etc.
- However, physical time of not necessarily what we need: only causality relationships between events often need to be preserved
- In this context, an event is typically the sending or the reception of some message

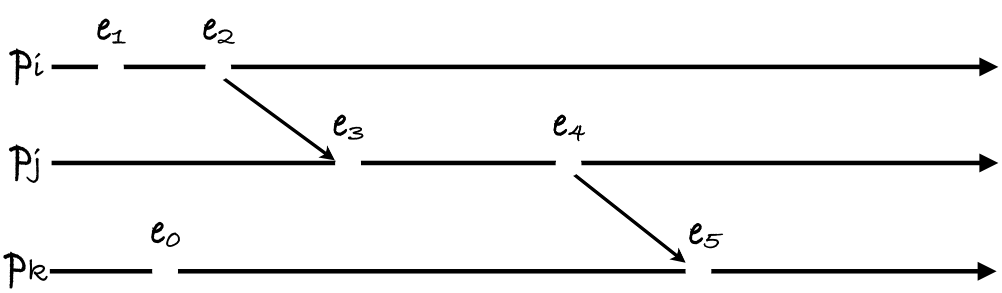
Causality relationship (1)

- In order to specify the causal broadcast, we must first introduce a partial order relationship
- Let $\rightarrow_\ell$ be a partial order on the set of events expressing direct dependencies such that:
  - Let e_1 and e_2 be two events occurring at the same process p: $e_1 \rightarrow_\ell e_2$ if and only if e_1 happened before e_2 at process p
  - In particular, we have that for each message m: send(m) $\rightarrow_\ell$ receive(m)

Causality relationship (2)

- We now define the causal ordering relationship, noted $\rightarrow_c$, as the transitive closure of $\rightarrow_\ell$
- Note that $\rightarrow_c$ also defines a partial order and is sometimes called the happened-before relationship
- Let e_1 and e_2 be two events occurring anywhere in the system, i.e., possibly at two distinct processes, we say that e_1 causally precedes e_2 if and only if we have $e_1 \rightarrow_c e_2$

Illustration of causality



- here we have $e_1 \rightarrow_c e_5$, via e_2, e_3 and e_4
- however, e_1 and e_0 are concurrent, i.e., they are not ordered (hence $\rightarrow_c$ is a partial order)

Causal broadcast (partial order)

We now specify causal broadcast by simply adding the causal order property given hereafter (based on the happened-before partial order) to the reliable broadcast properties

<u>Causal order</u>

If the broadcast of a message m <u>causally precedes</u> the broadcast of a message m', then no correct process delivers m' unless it has previously delivered m

So: <u>causal broadcast ⇔ reliable broadcast + causal order</u>

Causal broadcast (alternative)

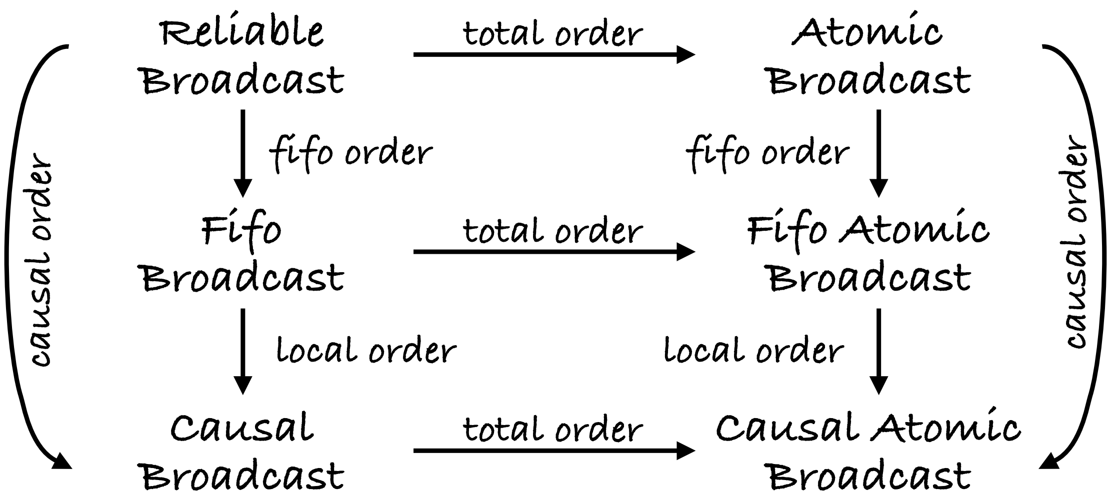
We can also see causal order as a <u>generalization of fifo order</u>. In this case, we define causal broadcast by adding the <u>local order</u> property given hereafter to the fifo broadcast properties

<u>Local order</u>

If a process broadcasts a message m and a process delivers m before broadcasting m', then no correct process delivers m' unless it has previously delivered m.

So: <u>causal broadcast ⇔ fifo broadcast + local order</u>

Relationship among broadcasts



Implementing broadcasts

- There exists numerous algorithms solving the various broadcast primitives we presented
- The algorithms we are presenting hereafter are taken from two major papers:

  [Hadzilacos93] Hadzilacos, V. and Toueg, S. 1993. *Fault-tolerant broadcasts and related problems*. In Distributed Systems (2nd Ed.), S. Mullender, Ed. Acm Press Frontier Series. ACM Press/Addison-Wesley Publishing Co., New York, NY, 97-145.

  [Chandra96] Chandra, T. D. and Toueg, S. 1996. *Unreliable failure detectors for reliable distributed systems*. J. ACM 43, 2 (Mar. 1996), 225-267.

- These algorithms all assume a partially synchronous system and might not be optimal

Reliable broadcast

Algorithm for process p:
To execute `broadcast`$(\texttt{R}, m)$:
    `send`(m) `to` p

`deliver`$(\texttt{R}, m)$ *occurs as follows:*
    upon `receive`(m) **do**
        if p has not previously executed `deliver`$(\texttt{R}, m)$ **then**
            `send`(m) `to` all neighbors
            `deliver`$(\texttt{R}, m)$

[Hadzilacos93]

Every process p *executes the following:*

To execute R-broadcast(m):
    send m to all (including p)

R-deliver(m) *occurs as follows:*
    when receive m for the first time
        if $sender(m) \neq p$ **then** send m to all
        R-deliver(m)

[Chandra96]

Comment: This is typically a flooding algorithm

Fifo broadcast

Algorithm for process p:
Initialization:
    $msgSet := \emptyset$
    $next[s] := 1$,   for each process s

To execute `broadcast`$(\texttt{F}, m)$:
    `broadcast`$(\texttt{R}, m)$

`deliver`$(\texttt{F}, -)$ *occurs as follows:*
    upon `deliver`$(\texttt{R}, m')$ **do**
        $s := sender(m')$
        if $next[s] = seq\#(m')$
        then
            `deliver`$(\texttt{F}, m')$
            $next[s] := next[s] + 1$
            while $(\exists m \in msgSet : sender(m) = s$
                and $next[s] = seq\#(m))$ **do**
                `deliver`$(\texttt{F}, m)$
                $next[s] := next[s] + 1$
        else
            $msgSet := msgSet \cup \{m'\}$

[Hadzilacos93]

Causal broadcast

Algorithm for process p:
Initialization:
$rcntDlvrs := \bot$
	`deliver(C,−)` *occurs as follows:*
	upon `deliver(F,` $\langle m_1, m_2, \ldots, m_l \rangle$`)` for some l **do**
	for $i := 1..l$ **do**
	if p has not previously executed `deliver(C,` m_i`)`
	then
	`deliver(C,` m_i`)`
	$rcntDlvrs := rcntDlvrs \,\|\|\, m_i$

To execute `broadcast(C,` m`)`: `broadcast(F,` $\langle rcntDlvrs \,\|\|\, m \rangle$`)` $rcntDlvrs := \bot$	

[Hadzilacos93]

Comments:

- rcntDelvrs is the sequence of messages that p delivered since it previous causal broadcast
- || is the concatenation operator on sequences of messages

Back to consensus...

The atomic broadcast can be reduced to the consensus problem. Note however that this version of consensus is different from the version we used when discussing atomic commitment. This second version is defined in terms of two primitives, propose(v) and decide(v), with v some value. When some process executes propose(v), we say that it proposes value v, and when it executes decides(v), we say it decices value v.

Termination. Every correct process eventually devices on some value.

Uniform integrity. Every process decides at most once.

Agreement. No two correct processes decide differently.

Uniform validity. If a process decides v, then v was proposed by some process.

Atomic broadcast

Initialization:
  $R\_delivered := \emptyset$
  $A\_delivered := \emptyset$
  $k := 0$

To execute `broadcast`(A, m):

  `broadcast`(R, m)

`deliver`$(A, -)$ occurs as follows:

  upon `deliver`(R, m) **do**
    $R\_delivered := R\_delivered \cup \{m\}$

  do forever
    $A\_undelivered := R\_delivered - A\_delivered$
    if $A\_undelivered \neq \emptyset$ **then**
      $k := k + 1$
      `propose`$(k, A\_undelivered)$
      wait for `decide`$(k, msgSet)$
      $batch(k) := msgSet - A\_delivered$
      A-deliver all messages in $batch(k)$ in some deterministic order
      $A\_delivered := A\_delivered \cup batch(k)$

[Hadzilacos93]

Initialisation:

  $R\_delivered \leftarrow \emptyset$
  $A\_delivered \leftarrow \emptyset$
  $k \leftarrow 0$

To execute *A-broadcast*(m):

  R-broadcast(m)

A-deliver$(-)$ occurs as follows:

  when *R-deliver*(m)
    $R\_delivered \leftarrow R\_delivered \cup \{m\}$

  when $R\_delivered - A\_delivered \neq \emptyset$
    $k \leftarrow k + 1$
    $A\_undelivered \leftarrow R\_delivered - A\_delivered$
    propose$(k, A\_undelivered)$
    wait until *decide*$(k, msgSet^k)$
    $A\_deliver^k \leftarrow msgSet^k - A\_delivered$
    atomically deliver all messages in $A\_deliver^k$ in some deterministic order
    $A\_delivered \leftarrow A\_delivered \cup A\_deliver^k$

[Chandra96]

Comment: consensus execution are numbered (k)

Questions?