Consensus Variants

Usman Mazhar Mirza

Consensus Variants

- In the variants we consider here, just like in consensus, the processes need to make consistent decisions, such as agreeing on one common value.
- Most of the abstractions extend or change the interface of consensus.

Consensus Variants

- Abstractions we will study are:
  - Total-Order Broadcast
  - Terminating Reliable Broadcast
  - Fast Consensus
  - Non-blocking Atomic Commitment
  - Group Membership
  - View Synchrony

We will mainly focus on **fail-stop algorithms** for implementing these abstractions. We will also consider **fail-arbitrary** model implementation for:

- Byzantine Total-Order Broadcast
- Byzantine Fast Consensus

Total-Order Broadcast: Overview

Earlier in Sect. 3.9, we discussed **FIFO-order** and **causal-order(reliable) broadcast** abstractions and their implementation.

FIFO-order broadcast requires that messages **from the same process** are delivered in the order that the sender has broadcast them. For messages **from different senders**, FIFO-order broadcast **does not guarantee** any particular order of delivery.

Causal-order broadcast **enforces a global ordering** for all messages that **causally depend on each other**: such messages need to be delivered in the same order and this order must respect causality. But causal-order broadcast **does not enforce any ordering** among messages that are **causally unrelated, or "concurrent"** in this sense.

Total-Order Broadcast: Overview

- A total-order (reliable) broadcast abstraction orders all messages, **even those from different senders and those that are not causally related.**
- More precisely, total order broadcast is a reliable broadcast communication abstraction which **ensures that all processes deliver the same messages in a common global order**.
- Whereas reliable broadcast ensures that processes agree on the **same set of messages** they deliver, total-order broadcast ensures that they agree on **the same sequence of messages**; the set of delivered messages is now ordered.

Total-Order Broadcast: Specifications

- We are considering two variants. The first is a **regular variant** that ensures total ordering only among the **correct processes**. The second is a **uniform variant** that ensures total ordering with respect to all processes, including the **faulty processes** as well.
- Total order property is **orthogonal** to the FIFO-order and causal-order properties. It is possible that a total-order broadcast abstraction does not respect causal order.

Total-Order Broadcast: Specifications

Module 6.1: Interface and properties of regular total-order broadcast

Module:

Name: TotalOrderBroadcast, **instance** *tob*.

First Variant: Total Order only among the correct processes

Events:

Request: $\langle$ *tob*, *Broadcast* | m $\rangle$: Broadcasts a message m to all processes.

Indication: $\langle$ *tob*, *Deliver* | p, m $\rangle$: Delivers a message m broadcast by process p.

Properties:

TOB1: *Validity:* If a correct process p broadcasts a message m, then p eventually delivers m.

TOB2: *No duplication:* No message is delivered more than once.

TOB3: *No creation:* If a process delivers a message m with sender s, then m was previously broadcast by process s.

TOB4: *Agreement:* If a message m is delivered by some correct process, then m is eventually delivered by every correct process.

Same as "reliable broadcast abstraction"

TOB5: *Total order:* Let m_1 and m_2 be any two messages and suppose p and q are any two correct processes that deliver m_1 and m_2. If p delivers m_1 before m_2, then q delivers m_1 before m_2.

Total-Order Broadcast: Specifications

Module 6.2: Interface and properties of uniform total-order broadcast

Module:

Name: UniformTotalOrderBroadcast, **instance** *utob*.

Second Variant: Total Order with respect to all processes

Events:

Request: ⟨ *utob*, *Broadcast* | m ⟩: Broadcasts a message m to all processes.

Indication: ⟨ *utob*, *Deliver* | p, m ⟩: Delivers a message m broadcast by process p.

Properties:

Same as "uniform reliable broadcast abstraction"

UTOB1–UTOB3: Same as properties TOB1–TOB3 in regular total-order broadcast (Module 6.1).

UTOB4: *Uniform agreement:* If a message m is delivered by some process (whether correct or faulty), then m is eventually delivered by every correct process.

UTOB5: *Uniform total order:* Let m_1 and m_2 be any two messages and suppose p and q are any two processes that deliver m_1 and m_2 (whether correct or faulty). If p delivers m_1 before m_2, then q delivers m_1 before m_2.

Fail-Silent Algorithm: Consensus-Based Total-Order Broadcast

- Implements the **first variant** of Total-Order broadcast abstraction.
- Uses **reliable broadcast abstraction** and **multiple instances** of (**regular**) consensus abstraction.
- Messages are first **disseminated** using a reliable broadcast instance. Recall that reliable broadcast imposes **no particular order** on delivering the messages, so every process simply stores the delivered messages in a **set of unordered messages**. At any point in time, it may be that no two processes have the same sets of unordered messages in their sets. The processes then use the **consensus** abstraction **to decide on one set**, order the messages in this set, and finally deliver them.

Algorithm 6.1: Consensus-Based Total-Order Broadcast

Implements:
TotalOrderBroadcast, **instance** *tob*.

Uses:
ReliableBroadcast, **instance** *rb*;
Consensus (multiple instances).

One consensus instance for every round

upon event $\langle$ *tob*, *Init* $\rangle$ **do**
unordered := $\emptyset$;
delivered := $\emptyset$;
round := 1;
wait := FALSE;

Wait flag to ensure that new round is not started before the previous round has terminated

upon event $\langle$ *tob*, *Broadcast* | m $\rangle$ **do**
trigger $\langle$ *rb*, *Broadcast* | m $\rangle$;

upon event $\langle$ *rb*, *Deliver* | p, m $\rangle$ **do**
if $m \notin$ *delivered* **then**
unordered := *unordered* $\cup \{(p, m)\}$;

upon *unordered* $\neq \emptyset \wedge$ *wait* = FALSE **do**
wait := TRUE;
Initialize a new instance *c.round* of consensus;
trigger $\langle$ *c.round*, *Propose* | *unordered* $\rangle$;

upon event $\langle$ *c.r*, *Decide* | *decided* $\rangle$ **such that** r = *round* **do**
forall $(s, m) \in$ sort(*decided*) **do** // by the order in the resulting sorted list
trigger $\langle$ *tob*, *Deliver* | s, m $\rangle$;
delivered := *delivered* $\cup$ *decided*;
unordered := *unordered* $\setminus$ *decided*;
round := *round* + 1;
wait := FALSE;

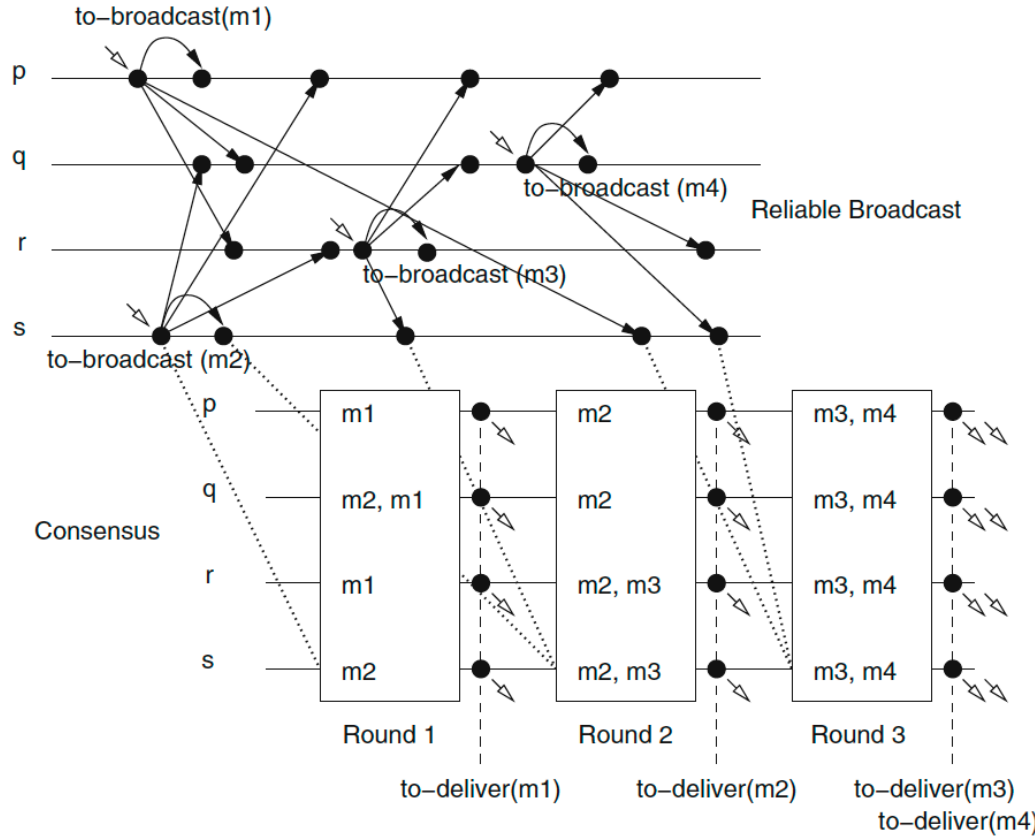

Figure 6.1: Sample execution of consensus-based total-order broadcast

Fail-Silent Algorithm: Consensus-Based Total-Order Broadcast

Consider the **total order property**. Let p and q be any two correct processes that to-deliver some message m**2**. Assume that p to-delivers some distinct message m**1** before m**2** . If p to-delivers m**1** and m**2** in the same round then due to the **agreement property of consensus**, q must have decided the same set of messages in that round. Thus, q also to-delivers m**1** before m**2**, as we assume that the messages decided in one round are to-delivered in the same order by every process, determined in a fixed way from the set of decided messages.

Byzantine Total-Order Broadcast: Overview

- Uses the same overall approach as the total-order broadcast abstraction with **crash-stop processes**.
- For implementing total-order broadcast in the **fail-arbitrary model**, however, one cannot simply take the algorithm from the fail-silent model and replace the underlying consensus primitive with Byzantine consensus.

Byzantine Total-Order Broadcast: Specifications

- The abstraction ensures the same **integrity property** as the Byzantine broadcast primitives in the sense that every message delivered with sender p was actually broadcast by p, if p is correct, and could not have been forged by Byzantine processes.
- Other properties are same as total-order broadcast among crash-stop processes.

Byzantine Total-Order Broadcast: Specifications

Module 6.3: Interface and properties of Byzantine total-order broadcast

Module:

Name: ByzantineTotalOrderBroadcast, **instance** *btob*.

Events:

Request: $\langle$ *btob*, *Broadcast* | m $\rangle$: Broadcasts a message m to all processes.

Indication: $\langle$ *btob*, *Deliver* | p, m $\rangle$: Delivers a message m broadcast by process p.

Properties:

BTOB1: *Validity:* If a correct process p broadcasts a message m, then p eventually delivers m.

BTOB2: *No duplication:* No correct process delivers the same message more than once.

BTOB3: *Integrity:* If some correct process delivers a message m with sender p and process p is correct, then m was previously broadcast by p.

BTOB4: *Agreement:* If a message m is delivered by some correct process, then m is eventually delivered by every correct process.

BTOB5: *Total order:* Let m_1 and m_2 be any two messages and suppose p and q are any two correct processes that deliver m_1 and m_2. If p delivers m_1 before m_2, then q delivers m_1 before m_2.

Fail-Noisy-Arbitrary Algorithm: Rotating Sender Byzantine Broadcast

- Byzantine broadcast abstractions are more complex because there are **no** useful **failure detector** abstractions.
- But an algorithm may rely on **eventual leader detector** primitive that is usually accessed through an **underlying consensus abstraction**.

Algorithm 6.2: Rotating Sender Byzantine Broadcast

Implements:
ByzantineTotalOrderBroadcast, **instance** *btob*.

Uses:
AuthPerfectPointToPointLinks, **instance** *al*;
ByzantineConsensus (multiple instances).

upon event ⟨ *btob*, *Init* ⟩ **do**
unordered := $([])^N$;
delivered := $\emptyset$;
round := 1;
wait := FALSE;
lsn := 0;
next := $[1]^N$;

upon event ⟨ *btob*, *Broadcast* | m ⟩ **do**
lsn := *lsn* + 1;
forall $q \in \Pi$ **do**
trigger ⟨ *al*, *Send* | q, [DATA, *lsn*, m] ⟩;

Each process send on authenticated links with sequence number

upon event ⟨ *al*, *Deliver* | p, [DATA, sn, m] ⟩ **such that** $sn = next[p]$ **do**
$next[p] := next[p] + 1$;
if $m \notin delivered$ **then**
$append(unordered[p], m)$;

upon exists p such that *unordered*[p] $\neq$ [] $\wedge$ *wait* $=$ FALSE **do**

wait := TRUE;

Initialize a new instance *bc.round* of Byzantine consensus;

if *unordered*[leader(*round*)] $\neq$ [] **then**

m := *head*(*unordered*[leader(*round*)]); **Returns first element**

else

$m := \square$; **Propose if process finds no message in the queue of process s.**

trigger ⟨ *bc.round*, *Propose* | m ⟩;

upon event ⟨ *bc.r*, *Decide* | m ⟩ **such that** r $=$ *round* **do**

s := *leader*(*round*);

if $m \neq \square \wedge m \notin$ *delivered* **then**

delivered := *delivered* $\cup$ $\{m\}$;

trigger ⟨ *btob*, Deliver | s, m ⟩;

remove(*unordered*[s], m);

round := *round* $+ 1$;

wait := FALSE;

Terminating Reliable Broadcast

- Reliable broadcast abstraction ensures that if a message is **delivered to a process** then it is **delivered to all correct processes** (in the uniform variant).

- Terminating reliable broadcast (TRB) is a form of reliable broadcast with a specific **termination property**. It is used in situations where a given process s is known to have the obligation of broadcasting some message to all processes in the system.

Terminating Reliable Broadcast

- Consider the case where process s crashes and some other process p detects that s has **crashed without having seen m**. It is possible that s crashed while broadcasting m. In fact, some processes might have delivered m whereas others might never do so. This can be problematic for an application.
- Process p might need to know **whether it should keep on waiting for m**, or if it can know at some point that m will never be delivered by any process.

Terminating Reliable Broadcast

Process p in the example **cannot decide** that it should wait for m or not. The TRB abstraction **adds precisely this missing piece of information** to reliable broadcast. TRB ensures that every process p either delivers the message m from the sender or some failure indication Δ, denoting that m will never be delivered (by any process).

Terminating Reliable Broadcast: Specifications

- The abstraction is defined for a specific sender process **s** , which is known to all processes in advance.
- Only the **sender process** broadcasts a message; all other processes invoke the algorithm and participate in the TRB upon initialization of the instance.
- The processes may not only deliver a message m but also "deliver" the special symbol Δ, which indicates that the sender has crashed.

Module 6.4: Interface and properties of uniform terminating reliable broadcast

Module:

Name: UniformTerminatingReliableBroadcast, **instance** *utrb*, with sender s.

Events:

Request: ⟨ *utrb*, *Broadcast* | m ⟩: Broadcasts a message m to all processes. Executed only by process s.

Indication: ⟨ *utrb*, *Deliver* | p, m ⟩: Delivers a message m broadcast by process p or the symbol $\triangle$.

Properties:

UTRB1: *Validity:* If a correct process p broadcasts a message m, then p eventually delivers m.

UTRB2: *Termination:* No process delivers more than one message.

UTRB3: *Integrity:* If a correct process delivers some message m, then m was either previously broadcast by process s or it holds $m = \triangle$.

UTRB4: *Uniform Agreement:* If any process delivers a message m, then every correct process eventually delivers m.

Fail-Stop: Consensus-Based Uniform Terminating Reliable Broadcast

- The sender process s disseminate a message m to all processes using best-effort broadcast. Every process waits until it **either receives the message** broadcast by the sender process or **detects the crash of the sender**.
- The properties of a perfect failure detector and the validity property of the broadcast ensure that **no process waits forever**. If the sender crashes, some processes may beb-deliver m and others may not beb-deliver any message.

Fail-Stop: Consensus-Based Uniform Terminating Reliable Broadcast

- Then all processes invoke the uniform consensus abstraction to agree on **whether to deliver m or the failure notification**.
- Every process **proposes either m or Δ** in the consensus instance, depending on whether the process has delivered m (from the best-effort broadcast primitive) or has detected the crash of the sender (in the failure detector).
- The decision of the consensus abstraction is then delivered by the algorithm. Note that, if a process has not beb-delivered any message from s then it learns m from the output of the consensus primitive.

Fail-Stop: Consensus-Based Uniform Terminating Reliable Broadcast

Algorithm 6.3: Consensus-Based Uniform Terminating Reliable Broadcast

Implements:
UniformTerminatingReliableBroadcast, **instance** *utrb*, with sender s.

Uses:
BestEffortBroadcast, **instance** *beb*;
UniformConsensus, **instance** *uc*;
PerfectFailureDetector, **instance** $\mathcal{P}$.

upon event ⟨ *utrb*, *Init* ⟩ **do**
 proposal := ⊥;

upon event ⟨ *utrb*, *Broadcast* | m ⟩ **do** // only process s
 trigger ⟨ *beb*, *Broadcast* | m ⟩;

upon event ⟨ *beb*, *Deliver* | s, m ⟩ **do**
 if *proposal* = ⊥ **then**
  proposal := m;
  trigger ⟨ *uc*, *Propose* | *proposal* ⟩;

Either "m" or "Δ" is proposed

upon event ⟨ $\mathcal{P}$, *Crash* | p ⟩ **do**
 if $p = s \wedge$ *proposal* = ⊥ **then**
  proposal := △;
  trigger ⟨ *uc*, *Propose* | *proposal* ⟩;

upon event ⟨ *uc*, *Decide* | *decided* ⟩ **do**
 trigger ⟨ *utrb*, *Deliver* | s, *decided* ⟩;

Example

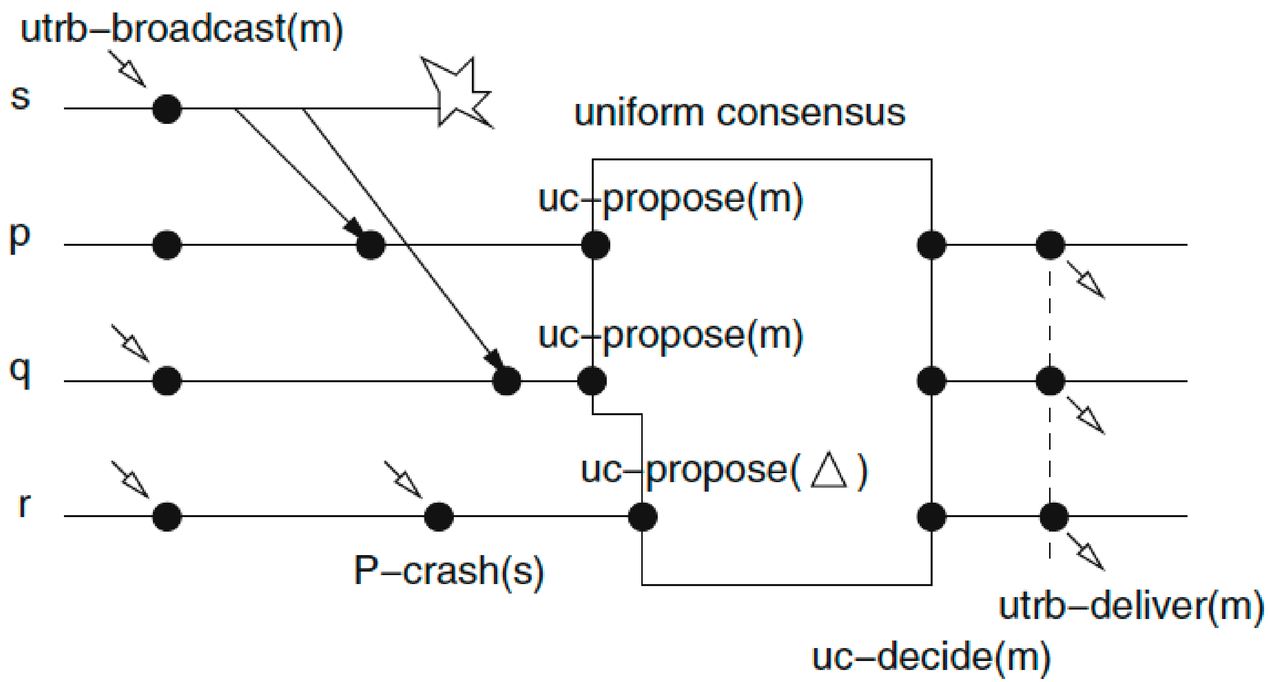

Figure 6.2: Sample execution of consensus-based uniform terminating reliable broadcast

Fast Consensus

- A consensus algorithm with good performance directly **accelerates** many implementations of other tasks as well.
- Many consensus algorithms **invoke multiple communication steps** with rounds of message exchanges among all processes.
- But some of these **communication steps may appear redundant**, especially for situations in which **all processes** start with the **same proposal value**.
- If the processes had a simple way to **detect that their proposals are the same**, consensus could be reached faster.

Fast Consensus

- Fast consensus is the variation of the consensus primitive with a requirement **to terminate** particularly fast under favorable circumstances. A fast consensus abstraction is a specialization of the consensus abstraction that must terminate in **one round** when all processes propose the **same value**.

Fast Consensus: Specifications

Module 6.5: Interface and properties of uniform fast consensus

Module:

Name: UniformFastConsensus, **instance** *ufc*.

Events:

Request: $\langle$ *ufc*, *Propose* | v $\rangle$: Proposes value v for consensus.

Indication: $\langle$ *ufc*, *Decide* | v $\rangle$: Outputs a decided value v of consensus.

Properties:

different

UFC1: *Fast termination:* If all processes propose the same value then every correct process decides some value after one communication step. Otherwise, every correct process eventually decides some value.

Same as uniform consensus

UFC2: *Validity:* If a process decides v, then v was proposed by some process.

UFC3: *Integrity:* No process decides twice.

UFC4: *Uniform agreement:* No two processes decide differently.

From Uniform Consensus to Uniform Fast Consensus

- It is a fail-silent algorithm and comes at the cost of **reduced resilience**. Specifically, implementing fast consensus requires that **N>3f instead of only N>2f**.
- Every process broadcasts its proposal value with best-effort guarantees. When a process receives only messages with the **same proposal value v** in this round, from N – f distinct processes, it decides v. This step ensures the fast termination property.

From Uniform Consensus to Uniform Fast Consensus

- Otherwise, if the messages received in the first round contain multiple distinct values, but still more than **N – 2f messages** contain the same proposal value w, the process adopts w as its own proposal value. **Unless the process has already decided, it then invokes an underlying uniform consensus primitive** with its proposal and lets it agree on a decision.

Algorithm 6.4: From Uniform Consensus to Uniform Fast Consensus

Implements:
    UniformFastConsensus, **instance** *ufc*.

Uses:
    BestEffortBroadcast, **instance** *beb*;
    UniformReliableBroadcast, **instance** *urb*;
    UniformConsensus, **instance** *uc*.

upon event ⟨ *uc*, *Init* ⟩ **do**
    proposal := ⊥;
    decision := ⊥;
    val := $[\perp]^N$;

upon event ⟨ *ufc*, *Propose* | *v* ⟩ **do**
    proposal := *v*;
    trigger ⟨ *beb*, *Broadcast* | [PROPOSAL, *proposal*] ⟩;

upon event $\langle$ *beb, Deliver* $\mid p$, [PROPOSAL, v] $\rangle$ **do**

  $val[p] := v$;   No decision has been made yet

  if $\#(val) = N - f \wedge decision = \bot$ **then**

    if exists $v \neq \bot$ such that $\#(\{p \in \Pi \mid val[p] = v\}) = N - f$ **then**

      $decision := v$;

      trigger $\langle$ *ufc, Decide* $\mid v \rangle$;

      trigger $\langle$ *urb, Broadcast* $\mid$ [DECIDED, *decision*] $\rangle$;

    else

      if exists $v \neq \bot$ such that $\#(\{p \in \Pi \mid val[p] = v\}) \geq N - 2f$ **then**

        $proposal := v$;

      $val := [\bot]^N$;

      trigger $\langle$ *uc, Propose* $\mid$ *proposal* $\rangle$;

upon event $\langle$ *urb, Deliver* $\mid p$, [DECIDED, v] $\rangle$ **do**

  if $decision = \bot$ **then**

    $decision := v$;

    trigger $\langle$ *ufc, Decide* $\mid v \rangle$;

upon event $\langle$ *uc, Decide* $\mid v \rangle$ **do**

  if $decision = \bot$ **then**

    $decision := v$;

    trigger $\langle$ *ufc, Decide* $\mid v \rangle$;

Thank you!