Consensus Variants

Usman Mazhar Mirza

Consensus Variants

- Some more abstractions:
  - Total-Order Broadcast
  - Terminating Reliable Broadcast
  - Fast Consensus
  - Group Membership
  - View Synchrony
  - Byzantine Total-Order Broadcast
  - Byzantine Fast Consensus

Fast Byzantine Consensus

- Fast consensus **decides** in **one round** whenever **all correct processes** propose the **same value**.

- Byzantine processes might propose **arbitrary** values, and a correct process **cannot distinguish** such a value from a value proposed by a correct process.

- Hence, one **cannot require** that an algorithm always decides in one round as **compared** to fast consensus with **crash-stop** process abstraction.

Fast Byzantine Consensus

- Fast Byzantine consensus abstraction requires a fast decision only in executions where **all processes are correct** (similar to weak byzantine consensus).

- Compared to the previous abstractions of Byzantine consensus, **deciding fast** in executions with unanimous proposals requires to **lower** the **resilience**. Specifically, the algorithm presented here assumes that **N > 5f**.

Fast Byzantine Consensus: Specification

- The primitive corresponds to a (**strong**) Byzantine consensus primitive with the **strengthened fast termination property** (properties FBC2–FBC4 are the **same** as properties BC2–BC4).

- The fast termination property requires that any fast Byzantine consensus algorithm **terminates after one communication step** if all correct processes propose the **same value**, but only in failure-free executions (**without Byzantine processes**).

Module 6.6: Interface and properties of fast Byzantine consensus

Module:

Name: FastByzantineConsensus, **instance** *fbc*.

Events:

Request: ⟨ *fbc*, *Propose* | v ⟩: Proposes value v for consensus.

Indication: ⟨ *fbc*, *Decide* | v ⟩: Outputs a decided value v of consensus.

Properties:

FBC1: *Fast termination:* If all processes are correct and propose the same value, then every correct process decides some value after one communication step. Otherwise, every correct process eventually decides some value.

FBC2: *Strong validity:* If all correct processes propose the same value v, then no correct process decides a value different from v; otherwise, a correct process may only decide a value that was proposed by some correct process or the special value $\square$.

FBC3: *Integrity:* No correct process decides twice.

FBC4: *Agreement:* No two correct processes decide differently.

Same as Byzantine Consensus Primitive

Fail-Arbitrary Algorithm:

- Transformation “From Byzantine Consensus to Fast Byzantine Consensus”.

- Same as Fast consensus but require some changes such as **lower resilience** (N>5f) and the **higher numbers of equal values** that are needed to decide fast.

- Every process **always invokes** the underlying (strong) Byzantine consensus primitive **even** after deciding fast. It simplifies the algorithm and avoids complications arising when broadcasting a decision value with byzantine processes.

Algorithm 6.5: From Byzantine Consensus to Fast Byzantine Consensus

Implements:

FastByzantineConsensus, **instance** *fbc*.

Uses:

AuthPerfectPointToPointLinks, **instance** *al*;
ByzantineConsensus, **instance** *bc*.

upon event $\langle$ *fbc*, *Init* $\rangle$ **do**
    $proposal := \square$;
    $decision := \bot$;
    $val := [\bot]^N$;

upon event $\langle$ *fbc*, *Propose* | v $\rangle$ **do**
    $proposal := v$;
    forall $q \in \Pi$ **do**
        trigger $\langle$ *al*, *Send* | q, [PROPOSAL, *proposal*] $\rangle$;

upon event $\langle$ *al*, *Deliver* | p, [PROPOSAL, v] $\rangle$ **do**
    $val[p] := v$;
    if $\#(val) = N - f$ **then**
        if exists $v \neq \bot$ such that $\#\big(\{p \in \Pi \mid val[p] = v\}\big) = N - f$ **then**
            $decision := v$;
            trigger $\langle$ *fbc*, *Decide* | v $\rangle$;
        if exists $v \neq \bot$ such that $\#\big(\{p \in \Pi \mid val[p] = v\}\big) > \frac{N-f}{2}$ **then**
            $proposal := v$;
        $val := [\bot]^N$;
        trigger $\langle$ *bc*, *Propose* | *proposal* $\rangle$;

upon event $\langle$ *bc*, *Decide* | v $\rangle$ **do**
    if $decision = \bot$ **then**
        $decision := v$;
        trigger $\langle$ *fbc*, *Decide* | v $\rangle$;

Protects a process from deciding more than once

Group Membership

- Different processes may get **notifications** (from failure detectors) about process **failures in different orders** and, in this way, obtain a different perspective of the system's evolution.
- Better coordinated failure notifications might result in **faster and simpler** algorithms.

- Group membership (GM) abstraction provides **consistent** and **accurate** information about which processes have **crashed** and which processes are **correct**.
- The output of a group membership primitive is better coordinated and at a **higher abstraction level** than the outputs of failure detectors and leader election modules.

Group Membership

- Membership abstraction also enables **dynamic changes** in the group of processes that constitute the system (No more static set of processes). Processes can **join and leave** the system.

- Group membership primitive also **coordinates** join and leave operations. As with failure notifications, it is desirable that group-membership information is provided to the processes in a consistent way.

Group Membership

- Assumptions (for simplicity):
  - Initial membership of the group is the complete set of processes.
  - Subsequent membership changes are solely caused by crashes.
  - No explicit join and leave operations.

Group Membership: Specification

- A group is the set of processes that participate in the computation.
- At any point in time, the current membership of the group is called the **group view**, or simply the **view**.
- A view V = (id, M) is a tuple where **id** is a unique numeric identifier and **M** is a set of view member processes. Overtime, the system may evolve through multiple views.
- The **initial group view** is the entire system, denoted by **V**0 = (0,**Π**).

Module 6.8: Interface and properties of group membership

Module:

Name: GroupMembership, **instance** *gm*.

Events: No request events to layer above.

Indication: $\langle$ *gm*, *View* | V $\rangle$: Installs a new view $V = (id, M)$ with view identifier id and membership M.

Properties: All correct processes install multiple new views in a sequence with monotonically increasing view identifiers

GM1: *Monotonicity:* If a process p installs a view $V = (id, M)$ and subsequently installs a view $V' = (id', M')$, then $id < id'$ and $M \supseteq M'$.

GM2: *Uniform Agreement:* If some process installs a view $V = (id, M)$ and another process installs some view $V' = (id, M')$, then $M = M'$.

Same as perfect failure detector

GM3: *Completeness:* If a process p crashes, then eventually every correct process installs a view (id, M) such that $p \notin M$.

GM4: *Accuracy:* If some process installs a view (id, M) with $q \notin M$ for some process $q \in \Pi$, then q has crashed.

Consensus-Based Group Membership

- After initialization, the algorithm **remains idle** until some process **detects** that another process has crashed.

- As different processes may detect crashes in **different orders**, a new view cannot be output immediately after detecting a failure and in a unilateral way; the processes first need to coordinate about the composition of the new view.

Algorithm 6.7: Consensus-Based Group Membership

Implements:

GroupMembership, **instance** *gm*.

Uses:

UniformConsensus (multiple instance);
PerfectFailureDetector, **instance** $\mathcal{P}$.

upon event $\langle$ *gm*, *Init* $\rangle$ **do**
$(id, M) := (0, \Pi)$;
correct := Π;
wait := FALSE;
trigger $\langle$ *gm*, *View* | (id, M) $\rangle$;

Installs initial view (each process)

Wait flag for preventing new consensus instance

upon event $\langle$ $\mathcal{P}$, *Crash* | *p* $\rangle$ **do**
correct := *correct* $\setminus$ $\{p\}$;

upon *correct* $\subsetneq M \wedge$ *wait* = FALSE **do**
$id := id + 1$;
wait := TRUE;
Initialize a new instance *uc.id* of uniform consensus;
trigger $\langle$ *uc.id*, *Propose* | *correct* $\rangle$;

upon event $\langle$ *uc.i*, *Decide* | M' $\rangle$ **such that** $i = id$ **do**
$M := M'$;
wait := FALSE;
trigger $\langle$ *gm*, *View* | (id, M) $\rangle$;

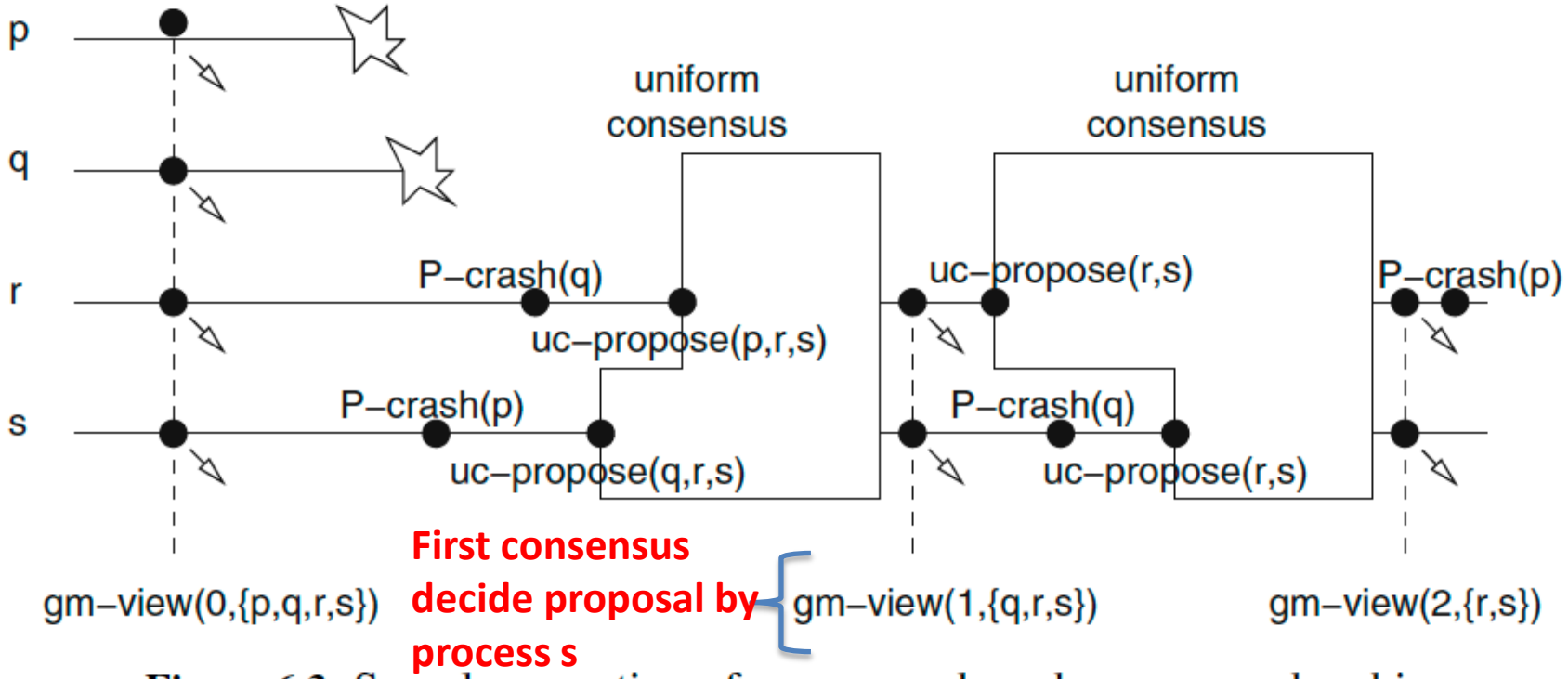

Figure 6.3: Sample execution of consensus-based group membership

View-Synchronous Communication/Broadcast Abstraction

- Integration of Reliable Broadcast and Group Membership.
- **Problem:**
  - In a group of processes, say, **process q , crashes**.
  - This failure is **detected** and membership abstraction installs a new view **V** = (**id, M**) at the processes such that **q is not in M**.
  - Suppose after **V** has been installed, some process **p** delivers a message **m** that was originally broadcasted by **q**.
  - Its desirable for p to simply **discard m**.

View-Synchronous Communication/Broadcast Abstraction

- **Problem Cont.:**
  - It may also happen that some other process **r** has **already delivered** **m** before installing view V.

- **Two conflicting goals:**
  - 1: To ensure the reliability of the broadcast, which means that **m** **must be delivered** by **p**.
  - 2: To guarantee the consistency of the view information, which means that **m** **cannot be delivered in the new view** and **p** must discard it.

View-Synchronous Communication/Broadcast Abstraction

- Solution **integrates** the installation of views with the delivery of messages and **orders** every new view with respect to the message flow.

- If a message **m** is delivered by a (**correct**) process **before it installs a view V** then **m** should be delivered by all processes that install V, before they install the view.

Specification

- "**View Inclusive Property**": A process delivers or broadcasts a message **m** in a view **V** if the process delivers or broadcasts **m**, respectively, after installing view **V** and before installing any subsequent view.

- The view inclusion property requires that every message must be delivered only in the same view in that it was broadcast.

Specification

- If new messages are continuously broadcast then the installation of a new view may be **postponed indefinitely**.
- Control on **broadcast pattern** is required.

- Solution is two events (Block & BlockOK) to handle communication with upper layer.

Module 6.9: Interface and properties of view-synchronous communication

Module:

Name: ViewSynchronousCommunication, **instance** *vs*.

Events:

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

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

Indication: ⟨ *vs*, *View* | V ⟩: Installs a new view $V = (id, M)$ with view identifier id and membership M.

Indication: ⟨ *vs*, *Block* ⟩: Requests that no new messages are broadcast temporarily until the next view is installed.

Request: ⟨ *vs*, *BlockOk* ⟩: Confirms that no new messages will be broadcast until the next view is installed.

Properties:

VS1: *View Inclusion:* If some process delivers a message m from process p in view V, then m was broadcast by p in view V.

VS2–VS5: Same as properties RB1–RB4 in (regular) reliable broadcast (Module 3.2).

VS6–VS9: Same as properties GM1–GM4 in group membership (Module 6.8).

Group Membership (Uniform)

Regular Broadcast Abstraction

Module 6.10: Interface and properties of uniform view-synchronous communication

Module:

Name: UniformViewSynchronousCommunication, **instance** *uvs*.

Events:

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

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

Indication: ⟨ *uvs*, *View* | V ⟩: Installs a new view $V = (id, M)$ with view identifier id and membership M.

Indication: ⟨ *uvs*, *Block* ⟩: Requests that no new messages are broadcast temporarily until the next view is installed.

Request: ⟨ *uvs*, *BlockOk* ⟩: Confirms that no new messages will be broadcast until the next view is installed.

Properties:

UVS1–UVS4 and **UVS6–UVS9:** Same as properties VS1–VS4 and VS6–VS9 in view-synchronous communication (Module 6.9).

UVS5: Same as property URB4 (*uniform agreement*) in uniform reliable broadcast (Module 3.3).

Uniform Reliable Broadcast Abstraction

Fail-Stop Algorithm: TRB-Based View-Synchronous Communication

- The keys element of the algorithm is a **collective flush procedure**, executed by the processes after they receive a view change from the underlying group membership primitive and before they install this new view at the view-synchronous communication level.

- During this step, every process uses an instance of the uniform TRB primitive to **rebroadcast** all messages that it has view-synchronously delivered in the current view.

Algorithm 6.8: TRB-Based View-Synchronous Communication (part 1, data transmission)

Implements:

ViewSynchronousCommunication, **instance** *vs*.

Uses:

UniformTerminatingReliableBroadcast (multiple instances);
GroupMembership, **instance** *gm*;
BestEffortBroadcast, **instance** *beb*.

upon event $\langle$ *vs*, *Init* $\rangle$ **do**
    $(vid, M) := (0, \emptyset)$; // current view $V = (vid, M)$
    flushing := FALSE; *blocked* := TRUE;
    inview := $\emptyset$;
    delivered := $\emptyset$;
    pendingviews := [];
    trbdone := $\emptyset$;

inview set is for each process with all sender/message pairs in current view

upon event $\langle$ *vs*, *Broadcast* | m $\rangle$ **such that** *blocked* = FALSE **do**
    inview := *inview* $\cup$ $\{(self, m)\}$;
    delivered := *delivered* $\cup$ $\{m\}$;
    trigger $\langle$ *vs*, *Deliver* | *self*, m $\rangle$;
    trigger $\langle$ *beb*, *Broadcast* | [DATA, *vid*, m] $\rangle$;

upon event $\langle$ *beb*, *Deliver* | p, [DATA, *id*, m] $\rangle$ **do**
    if $id = vid \wedge blocked =$ FALSE $\wedge\ m \notin delivered$ **then**
        inview := *inview* $\cup$ $\{(p, m)\}$;
        delivered := *delivered* $\cup$ $\{m\}$;
        trigger $\langle$ *vs*, *Deliver* | p, m $\rangle$;

Algorithm 6.9: TRB-Based View-Synchronous Communication (part 2, view change)

upon event $\langle$ *gm*, *View* | V' $\rangle$ **do**
    if $V' = (0, M')$ for some M' **then** // start the initial view
        $(vid, M) := (0, M')$;
        blocked := FALSE;
    else
        append(*pendingviews*, V');

upon *pendingviews* $\neq$ [] $\wedge$ *flushing* = FALSE **do** // start collective flush procedure
    flushing := TRUE;
    trigger $\langle$ *vs*, *Block* $\rangle$;

upon event $\langle$ *vs*, *BlockOk* $\rangle$ **do**
    blocked := TRUE;
    forall $p \in M$ **do**
        Initialize a new instance *utrb.vid.p* of uniform terminating reliable
            broadcast with sender p;
        if $p = self$ **then**
            trigger $\langle$ *utrb.vid.p*, *Broadcast* | *inview* $\rangle$;

upon event $\langle$ *utrb.id.p*, *Deliver* | p, iv $\rangle$ **such that** $id = vid$ **do**
    trbdone := *trbdone* $\cup$ $\{p\}$;
    if $iv \neq \triangle$ **then**
        forall $(s, m) \in iv$ such that $m \notin$ *delivered* **do**
            delivered := *delivered* $\cup$ $\{m\}$;
            trigger $\langle$ *vs*, *Deliver* | s, m $\rangle$;

upon *trbdone* = $M \wedge$ *blocked* = TRUE **do** // ready to install new view
    V := *head*(*pendingviews*); *remove*(*pendingviews*, V);
    $(vid, M) := V$;
    correct := *correct* $\cap$ M;
    flushing := FALSE; *blocked* := FALSE;
    inview := $\emptyset$;
    trbdone := $\emptyset$;
    trigger $\langle$ *vs*, *View* | (vid, M) $\rangle$;

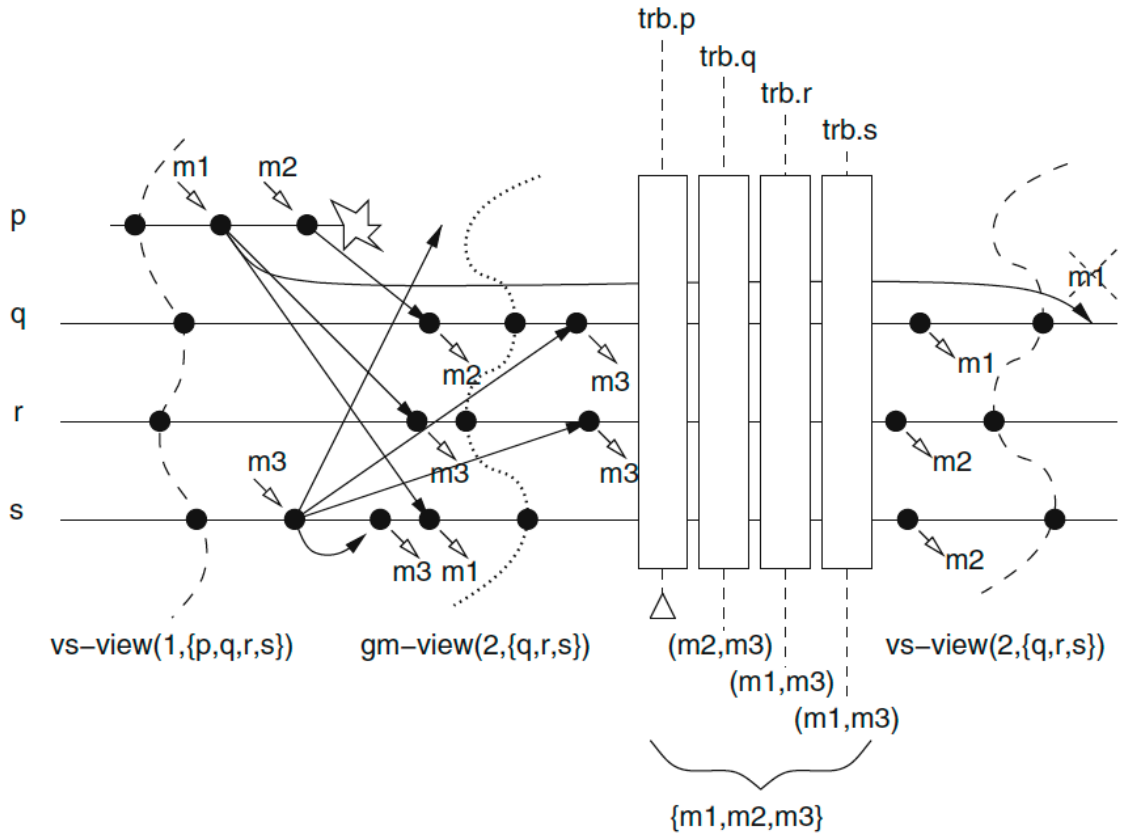

Figure 6.4: Sample execution of the TRB-based view synchronous algorithm

Consensus-Based Uniform View-Synchronous Communication

- Previous Algorithm is uniform in the sense that no two processes, be they correct or not, install different views.
- The algorithm is **not uniform** in the message-delivery sense, as required by uniform view-synchronous communication (using best effort broadcast).

Algorithm 6.10: Consensus-Based Uniform View-Synchronous Communication (part 1)

Implements:
UniformViewSynchronousCommunication, **instance** *uvs*.

Uses:
UniformConsensus (multiple instances);
BestEffortBroadcast, **instance** *beb*;
PerfectFailureDetector, **instance** $\mathcal{P}$.

No TRB or Group Membership

upon event $\langle$ *uvs*, *Init* $\rangle$ **do**
  $(vid, M) := (0, \Pi)$; // current view $V = (vid, M)$
  $correct := \Pi$;
  flushing := FALSE; *blocked* := FALSE; *wait* := FALSE;
  $pending := \emptyset$;
  $delivered := \emptyset$;
  forall m **do** $ack[m] := \emptyset$;
  $seen := [\bot]^N$;
  trigger $\langle$ *uvs*, *View* | (vid, M) $\rangle$;

upon event $\langle$ *uvs*, *Broadcast* | m $\rangle$ **such that** *blocked* = FALSE **do**
  $pending := pending \cup \{(self, m)\}$;
  trigger $\langle$ *beb*, *Broadcast* | [DATA, *vid*, *self*, m] $\rangle$;

upon event $\langle$ *beb*, *Deliver* | p, [DATA, *id*, s, m] $\rangle$ **do**
  if $id = vid \wedge blocked =$ FALSE **then**
    $ack[m] := ack[m] \cup \{p\}$;
    if $(s, m) \notin pending$ **then**
      $pending := pending \cup \{(s, m)\}$;
      trigger $\langle$ *beb*, *Broadcast* | [DATA, *vid*, s, m] $\rangle$;

upon exists $(s, m) \in pending$ such that $M \subseteq ack[m] \wedge m \notin delivered$ **do**
  $delivered := delivered \cup \{m\}$;
  trigger $\langle$ *uvs*, *Deliver* | s, m $\rangle$;

Algorithm 6.11: Consensus-Based Uniform View-Synchronous Communication (part 2)

upon event $\langle\ \mathcal{P}, Crash \mid p\ \rangle$ **do**
    $correct := correct \setminus \{p\}$;

upon $correct \supsetneq M \wedge flushing = \text{FALSE}$ **do**
    $flushing := \text{TRUE}$;
    trigger $\langle\ uvs, Block\ \rangle$;

upon event $\langle\ uvs, BlockOk\ \rangle$ **do**
    $blocked := \text{TRUE}$;
    trigger $\langle\ beb, Broadcast \mid [\text{PENDING}, vid, pending]\ \rangle$;

upon event $\langle\ beb, Deliver \mid p, [\text{PENDING}, id, pd]\ \rangle$ **such that** $id = vid$ **do**
    $seen[p] := pd$;

upon (forall $p \in correct : seen[p] \neq \perp$) $\wedge\ wait = \text{FALSE}$ **do**
    $wait := \text{TRUE}$;
    $vid := vid + 1$;
    Initialize a new instance $uc.vid$ of uniform consensus;
    trigger $\langle\ uc.vid, Propose \mid (correct, seen)\ \rangle$;

upon event $\langle\ uc.id, Decide \mid (M', S)\ \rangle$ **such that** $id = vid$ **do** // install new view
    forall $p \in M'$ such that $S[p] \neq \perp$ **do**
        forall $(s, m) \in S[p]$ such that $m \notin delivered$ **do**
            $delivered := delivered \cup \{m\}$;
            trigger $\langle\ uvs, Deliver \mid s, m\ \rangle$;
    $flushing := \text{FALSE}$; $blocked := \text{FALSE}$; $wait := \text{FALSE}$;
    $pending := \emptyset$;
    forall m **do** $ack[m] := \emptyset$;
    $seen := [\perp]^N$;
    $M := M'$;
    trigger $\langle\ uvs, View \mid (vid, M)\ \rangle$;

Thank you!