Group Membership

Who is there?
Group Membership

- In some distributed applications, processes need to know which processes are *participating* in the computation and which are not.
- Failure detectors provide such information; however, that information is *not coordinated* (see next slide) even if the failure detector is perfect.
Perfect Failure Detector

p1

p2

p3

p4

suspect(p2)  suspect(p2,p3)

crash

suspect(p3)  suspect(p2,p3)

suspect()
Group Membership

V1 = (p1, p4)

p1

crash

p2

crash

p2

crash

p4

V1 = (p1, p4)
Group Membership

• To illustrate the concept, we focus here on a group membership abstraction to coordinate the information about crashes.

• In general, a group membership abstraction can also typically be used to coordinate the processes joinning and leaving explicitly the set of processes (i.e., without crashes).
Group Membership

- **Like** with a failure detector, the processes are informed about failures; we say that the processes install views

- **Like** with a perfect failure detector, the processes have accurate knowledge about failures

- **Unlike** with a perfect failure detector, the information about failures are *coordinated*: the processes install the same sequence of views
**Group Membership**

**Memb1. Local Monotonicity:** If a process installs view 
(j,M) after installing (k,N), then j > k and M < N

**Memb2. Agreement:** No two processes install views (j,M)  
and (j,M’) such that M ≠ M’

**Memb3. Completeness:** If a process p crashes, then  
there is an integer j such that every correct process  
eventually installs view (j,M) such that p ∉ M

**Memb4. Accuracy:** If some process installs a view (i,M)  
and p ∉ M, then p has crashed
Group Membership

Events

Indication: <membView, V>

Properties:

- Memb1, Memb2, Memb3, Memb4
Algorithm (gmp)

- **Implements**: groupMembership (gmp).
- **Uses**:
  - PerfectFailureDetector (P).
  - UniformConsensus(Ucons).

**upon event < Init > do**

- view := (0,S);
- correct := S;
- wait := true;
Algorithm (gmp – cont’d)

- **upon event** `< crash, pi >` **do**
  - `correct := correct \ {pi};`

- **upon event** `(correct < view.memb) and (wait = false)` **do**
  - `wait := true;`
  - `trigger<ucPropose,(view.id+1,correct) >;`
Algorithm (gmp – cont’d)

• **upon event** < ucDecided, (id, memb)> **do**
  • view := (id, memb);
  • wait := false;
  • **trigger** < membView, view>;}
Algorithm (gmp)

\[
\text{UCons}((p1,p2,p4);(p1,p2,p4))
\]

\[
\text{crash}
\]

\[
\text{UCons}((p1,p3,p4);(p1,p2,p4))
\]

\[
\text{crash}
\]

\[
\text{UCons}((p1,p4);(p1,p4))
\]
Group Membership and Broadcast

membView(p1,p3)

p1

p2

p3

m

m

crash

membView(p1,p3)
View Synchrony

- **View synchronous broadcast** is an abstraction that results from the combination of group membership and reliable broadcast.

- **View synchronous broadcast** ensures that the delivery of messages is coordinated with the installation of views.
View Synchrony

Besides the properties of **group membership** *(Memb1-Memb4)* and **reliable broadcast** *(RB1-RB4)*, the following property is ensured:

**VS:** A message is **vsDelivered** in the view where it is **vsBroadcast**
View Synchrony

Events

Request:

<vsBroadcast, m>

Indication:

- <vsDeliver, src, m>
- <vsView, V>
View Synchrony

If the application keeps \texttt{vsBroadcasting} messages, the \textit{view synchrony} abstraction might never be able to \texttt{vsInstall} a new view; the abstraction would be impossible to implement.

We introduce a specific event for the abstraction to \texttt{block} the application from \texttt{vsBroadcasting} messages; this only happens when a process crashes.
View Synchrony

Events

Request:

<vsBroadcast, m>; <vsBlock, ok>

Indication:

<vsDeliver, src, m>; <vsView, V>; <vsBlock>
Algorithm (vsc)

- **Implements:** ViewSynchrony (vs).

- **Uses:**
  - GroupMembership (gmp).
  - TerminatingReliableBroadcast (trb).
  - BestEffortBroadcast (beb).
Algorithm (vsc – cont’d)

upon event < Init > do

  view := (0,S); nextView := ⊥;
  pending := delivered := trbDone := ∅;
  flushing := blocked := false;
Algorithm (vsc – cont’d)

Upon event <vsBroadcast,m> and (blocked = false) do

- delivered := delivered \cup \{ m \}
- trigger <vsDeliver, self, m>;
- trigger <bebBroadcast, [Data,view.id,m]>;
Algorithm (vsc – cont’d)

upon event<bebDeliver,src,[Data,vid,m]) do

If(view.id = vid) and (m \notin delivered) and
(blocked = false) then

delivered := delivered \cup \{ m \}

trigger <vsDeliver, src, m >;
Algorithm (vsc – cont’d)

upon event < membView, V > do
  addtoTail (pending, V);

upon (pending ≠ ∅) and (flushing = false) do
  nextView := removeFromhead (pending);
  flushing := true;
  trigger <vsBlock>;
Algorithm (vsc – cont’d)

Upon <vsBlockOk> do
  blocked := true;
  trbDone:= ∅;
  trigger <trbBroadcast, self, (view.id,delivered)>;
Algorithm (vsc – cont’d)

Upon <trbDeliver, p, (vid, del)> do

trbDone := trbDone ∪ \{ p \};

forall m ∈ del and m ∉ delivered do

delivered := delivered ∪ \{ m \};

trigger <vsDeliver, src, m >;
Algorithm (vsc – cont’d)

Upon (trbDone = view.memb) and (blocked = true) do

view := nextView;
flushing := blocked := false;
delivered := ∅;

trigger <vsView, view>;}
Consensus-Based View

Synchrony

Instead of launching parallel instances of TRBs, plus a group membership, we use one consensus instance and parallel broadcasts for every view change.

Roughly, the processes exchange the messages they have delivered when they detect a failure, and use consensus to agree on the membership and the message set.
Algorithm 2 (vsc)

- **Implements:** ViewSynchrony (vs).

  - **Uses:**
    - UniformConsensus (uc).
    - BestEffortBroadcast(beb).
    - PerfectFailureDetector(P).
Algorithm 2 (vsc – cont’d)

upon event < Init > do

view := (0,S);
correct := S;
flushing := blocked := false;
delivered := dset := ∅;
Algorithm 2 (vsc – cont’d)

upon event <vsBroadcast,m) and (blocked = false) do

  delivered := delivered \cup \{ m \}

  trigger <vsDeliver, self,m>;

  trigger <bebBroadcast,[Data,view.id,m] >;
Algorithm 2 (vsc – cont’d)

upon event<bebDeliver,src,[Data,vid,m]) do

if (view.id = vid) and m ∉ delivered and blocked = false then

delivered := delivered ∪ { m }

trigger <vsDeliver, src, m >;
Algorithm 2 (vsc – cont’d)

upon event < crash, p > do
  correct := correct \{ p \};
  if flushing = false then
    flushing := true;
    trigger <vsBlock>;
Algorithm 2 (vsc – cont’d)

Upon <vsBlockOk> do

   blocked := true;

   trigger <bebBroadcast, [DSET, view.id, delivered] >;
Algorithm 2 (vsc – cont’d)

Upon <bebDeliver, src, [DSET, vid, del] > do

\[ \text{dset} := \text{dset} \cup (\text{src, del}); \]

if forall \( p \in \text{correct}, (p, \text{mset}) \in \text{dset} \) then

trigger <ucPropose, view.id+1, correct, dset >;
Algorithm 2 (vsc – cont’d)

Upon \(<uc\text{Decided}, \text{id}, \text{memb}, \text{vsdset}>\) do

forall \((p, mset) \in \text{vsdset}: p \in \text{memb}\) do

forall \((\text{src}, m) \in \text{mset}: m \not\in \text{delivered}\) do

\hspace{1cm} \text{delivered} := \text{delivered} \cup \{m\}

\hspace{1cm} \text{trigger} <\text{vsDeliver}, \text{src}, m>;

view := (\text{id}, \text{memb}); \text{flushing} := \text{blocked} := \text{false}; \text{dset} := \text{delivered} := \emptyset;

\text{trigger} <\text{vsView}, \text{view}>;
Uniform View Synchrony

We now combine the properties of

*group membership* (**Memb1-Memb4**) – which is already uniform

*uniform reliable broadcast* (**RB1-RB4**) – which we require to be uniform

**VS:** A message is **vsDelivered** in the view where it is **vsBroadcast** – which is already uniform
Uniform View Synchrony

Using uniform reliable broadcast instead of best effort broadcast in the previous algorithms does not ensure the uniformity of the message delivery.
Uniformity?

vsView(p1,p3)

vsDeliver(m)

crash

vsView(p1,p3)
Algorithm 3 (uvsc)

upon event < Init > do

view := (0, S);
correct := S;
flushing := blocked := false;
udelivered := delivered := dset := ∅;
for all m: ack(m) := ∅;
Algorithm 3 (uvsc – cont’d)

 upon event <vsBroadcast,m) and (blocked = false) do
        delivered := delivered ∪ {m};
        trigger <bebBroadcast,[Data,view.id,m]>;
Algorithm 3 (uvsc – cont’d)

upon event <bebDeliver, src, [Data, vid, m]) do
    if (view.id = vid) then
        ack(m) := ack(m) ∪ {src};
    if m ∉ delivered then
        delivered := delivered ∪ {m}
    trigger <bebBroadcast, [Data, view.id, m]>;
Algorithm 3 (uvsc – cont’d)

upon event (view ≤ ack(m)) and (m ∉ udelivered)
do

udelivered := udelivered ∪ { m }

trigger <vsDeliver, src(m), m >;
Algorithm 3 (uvsc – cont’d)

```plaintext
upon event < crash, p > do
    correct := correct \{ p \};
    if flushing = false then
        flushing := true;
        trigger <vsBlock>;
```

Algorithm 3 (uvsc – cont’d)

Upon <vsBlockOk> do

  blocked := true;

  trigger <bebBroadcast, [DSET, view.id, delivered] >;

Upon <bebDeliver, src, [DSET, vid, del] > do

dset := dset \cup (src, del);

if forall p \in correct, (p, mset) \in dset
then trigger <ucPropose, view.id+1, correct, dset >;
Algorithm 3  (uvsc – cont’d)

Upon <ucDecided, id, memb, vsdset> do

forall (p,mset) ∈ vs-dset: p ∈ memb do

forall (src,m) ∈ mset: m ∉ udelivered do

udelivered := udelivered ∪ {m}

trigger <vsDeliver, src, m>;

view := (id, memb); flushing := blocked := false; dset := delivered := udelivered := ∅;

trigger <vsView, view>;}