Distributed Systems

Group Membership and View Synchronous Communication

Prof R. Guerraoui
Distributed Programming Laboratory
Group Membership

Who is there?
Group Membership

• In many 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

suspect(p2)  suspect(p2, p3)

crash

suspect()  suspect(p3)  suspect(p2, p3)

p1  p2  p3  p4
Group Membership

V1 = (p1, p4)

p1

p2

p2

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 *joining* 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 \(M \neq M'\)

**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 \(p \notin M\)

**Memb4. Accuracy:** If some process installs a view \((i,M)\) and \(p \notin M\), 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)

UCons((p1,p2,p4);(p1,p2,p4))

UCons((p1,p4);(p1,p4))

UCons((p1,p3,p4);(p1,p2,p4))

p1

p2

p3

p4
Group Membership and Broadcast

membView(p1,p3)

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:
  - $\langle \text{vsBroadcast}, \ m \rangle$

- Indication:
  - $\langle \text{vsDeliver}, \ \text{src}, \ m \rangle$
  - $\langle \text{vsView}, \ V \rangle$
View Synchrony

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

We introduce a specific event for the abstraction to $block$ the application from $vsBroadcasting$ 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)

\[\text{upon event} \ < \ \text{membView}, \ V \ > \ \text{do}\]
\begin{itemize}
\item \text{addtoTail} \ (\text{pending}, \ V);\end{itemize}

\[\text{upon} \ (\text{pending} \neq \emptyset) \ \text{and} \ (\text{flushing} = \text{false}) \ \text{do}\]
\begin{itemize}
\item nextView := \text{removeFromHead} \ (\text{pending});\end{itemize}
\begin{itemize}
\item flushing := true;\end{itemize}
\begin{itemize}
\item trigger <vsBlock>;</itemize>
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 >;
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 ∪ {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

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

if forall p \( \in \) correct, (p,mset) \( \in \) dset then
trigger <ucPropose, view.id+1, correct, dset >;
Algorithm 2 (vsc – cont’d)

Upon <ucDecided, id, memb, vsdset> do
forall (p,mset) ∈ vsdset: p ∈ memb do
forall (src,m) ∈ mset: m ∉ delivered do
  delivered := delivered ∪ {m}
trigger <vsDeliver, src, m>;
view := (id, memb); flushing := blocked := false; dset := delivered := ∅;
trigger <vsView, 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?

\[ \text{vsView}(p1,p3) \]

\[ \text{vsDeliver}(m) \]

\[ \text{crash} \]
Algorithm 3 (uvsc)

upon event < Init > do

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

upon event <vsBroadcast,m) and (blocked = false) do
delivered := delivered \cup \{m\};

trigger <bebBroadcast,[Data,view.id,m] >;
Algorithm 3 (uvsc – cont’d)

\begin{algorithm}
\begin{algorithmic}
\State \textbf{upon event} <bebDeliver,src,[Data,vid,m]) \textbf{do}
\State \hspace{1em} \textbf{if} (view.id = vid) \textbf{then}
\State \hspace{2em} ack(m) := ack(m) \cup \{ src \};
\State \hspace{1em} \textbf{if} m \notin \text{delivered} \textbf{then}
\State \hspace{2em} delivered := delivered \cup \{ m \}
\State \hspace{1em} \textbf{trigger} <bebBroadcast, [Data,view.id,m] >;
\end{algorithmic}
\end{algorithm}
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)

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 ∪ (src,del);
  if forall p ∈ correct, (p,mset) ∈ 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>;`