

6 View-synchronous Group Communication

6.1 Introduction

Consensus and reliable broadcast have been considered in *static* groups. Systems with *dynamic* groups extend this model by providing explicit *join* and *leave* operations to adapt the group membership over time. Moreover, such systems exclude failed servers automatically from the membership. Still, reaching agreement on the group membership in the presence of failures is not trivial.

Two approaches have been considered:

1. Run a consensus protocol among the all previous group members to agree on the future group membership. This is the canonical approach, tolerates further failures during the membership change, but involves the potentially expensive consensus primitive.
2. Integrate consensus with the membership protocol and run it only among the (hopefully) correct members. Since this consensus algorithm needs not tolerate failures, it can be simpler; but because further failures may still occur, it provides different guarantees.

The second approach is taken by *view-synchronous* group communication systems and related *group-membership* algorithms [Pow96].

The first view-synchronous group communication systems was ISIS [BJ87]; many more followed and have been used in real-world applications like trading floor communication for the stock market or air-traffic control systems. IBM's Reliable Scalable Cluster Technology (RSCT) is the failure-detection and group membership component in IBM's High-Availability Cluster Multiprocessing (HACMP) system for AIX, Linux, and Windows clusters (www-03.ibm.com/systems/power/software/availability/aix/index.html); The Distribution & Consistency Services (DCS) [IBM06], available with WebSphere Application Server Network Deployment from IBM (www-01.ibm.com/software/webervers/appserv/was/), is a group communication system for Java applications. Many other products, like large database systems or file servers, contain similar technology. The High Availability Linux Project (www.linux-ha.org), Spread (www.spread.org), and JGroups (www.jgroups.org) are well-known open-source high-availability toolkits and group communication systems.

6.2 Group membership

The system model is the same as in the chapter on failure detectors and reliable broadcast; in particular, every server P_i has access to a local failure detector that it uses to determine the group membership.

A *group-membership service* receives *join*(S) and *leave*(S) requests with $S \subset \mathcal{P}$ and runs a failure detector to discover faulty servers. It outputs a sequence of group membership sets that are called *views*. Every view $V \subseteq \mathcal{P}$ is delivered through a *v-change*(vid, V) event, where

$vid \in \mathbb{N}$ denotes a monotonically increasing *view identifier*. When this occurs, we say that the server (process) *installs* the new view V .

Apart from handling explicitly given join and leave operations, a membership service plays a dual role as a failure detector: it should remove crashed servers from the group and detect the “stable” components of the system, i.e., the set of servers who are correct and who can reliably communicate with each other. Formally, a *stable* component is defined as a set of servers that is eventually permanently connected.

Definition 1 (Membership service). A group-membership service satisfies:

Membership: If some correct server P_i processes a $join(S)$ or $leave(S)$ request and no server processes any further such requests, then P_i eventually installs a view V such that $S \subseteq V$ or $S \cup V = \emptyset$, respectively.

View agreement: If P_i installs a view V with view identifier vid and P_j installs a view V' with view identifier vid , then $V = V'$.

Self-inclusion: If P_i installs a view V , then $P_i \in V$.

Monotonicity: If P_i installs a view V with identifier vid after installing a view V' with identifier vid' , then $vid > vid'$.

Precision: For every *stable* component $S \subseteq \mathcal{P}$, the following holds:

- (i) every server $P_i \in S$ eventually installs S as its last view; and
- (ii) every message that P_i sends while in view S is received by every other server $P_j \in S$ while it is in view S .

A membership service can be implemented using an eventually perfect failure detector. In order to avoid problems with *monotonicity*, a server that crashes and recovers is usually given a new identity before it can rejoin the group.

6.3 View-synchronous broadcast

One of the most important goals of a group communication system is to implement reliable (FIFO, causally ordered, or atomic) broadcast. For this, view-synchrony guarantees the agreement property of reliable broadcast among the servers in a stable component. Furthermore, view-synchrony links the set of messages delivered within a view to the servers in the view and to the servers in the subsequent view. This is necessary in order to give a useful guarantee to an application at the start of a new view. Through this, the application knows who delivered the same messages in the past view and can create a consistent state among the members of the new view.

Formally, view-synchronous reliable broadcast is characterized by an event $v\text{-broadcast}(m)$ to broadcast a message m and by an event $v\text{-deliver}(m)$, through which a message m is received. The view-synchrony property is defined with respect to the sequence of views delivered by the group-membership service.

Definition 2 (View-synchronous reliable broadcast). A view-synchronous reliable broadcast protocol satisfies:

Same-view-delivery: If a server P_i v -broadcasts a message m in some view V and a server P_j v -delivers m in view V' , then $V = V'$.

View-synchrony: If two servers P_i and P_j both install a new view V in the same previous view V' , then any message v -delivered by P_i in V' was also v -delivered by P_j in V' .

Integrity: Every server delivers at most one message m , and only if m was previously broadcast by the associated sender.

The *view-synchrony* property implies that all servers who proceed together from one view V' through a view change to the next view V have v -delivered the same messages in V' . Therefore, no further synchronization among them is needed. Newly joining servers, i.e., servers in V that were not also in V' , need to receive the missing messages from a member of V' .

But *view-synchrony* says nothing about the messages that were delivered at servers which did not proceed together from the same view to the next. In order for a server to find out which others have the same state, additional information in a so-called *transitional set* is given in some group communication systems:

Transitional set: When a server P_i installs a view V in previous view V' , then it also delivers a *transitional set* $T_i \subseteq V \cap V'$ such that any P_j that also installs V is contained in T_i if and only if P_j 's previous view was also V' .

The following implementation of view-synchronous reliable broadcast blocks the application during view changes so that it may not v -broadcast any messages for some time. (Many practical systems have this property.) Messages between servers are sent over reliable point-to-point links with FIFO delivery. As mentioned above, the protocol also delivers the views received from the underlying group membership to the application.

Algorithm 3 (View-synchronous reliable broadcast). Algorithm for P_i :

initialization:

$s \leftarrow 0$ // P_i 's sequence number
 $s_j \leftarrow 0 \quad \forall j \in [1, n]$ // sequence number of last v -delivered message from P_j
 $vid \leftarrow 0; view \leftarrow \{P_i\}$ // current view
 $new_vid \leftarrow 0; new_view \leftarrow \emptyset$ // next view while it is being installed
 $delivered \leftarrow \emptyset; resent \leftarrow \emptyset$ // messages v -delivered and resent, resp., in current view

upon v -broadcast(m):

send message (`send`, vid , s , m) to all servers in $view$
 $s \leftarrow s + 1$

upon receiving a message¹ (`send`, vid , s' , m) from P_j :

$delivered \leftarrow delivered \cup \{j, s', m\}$
if $new_vid = 0$ **or** $P_j \in view \cap new_view$ **then**
 v -deliver(m)
 $s_j \leftarrow s'$

upon v -change(v, V):

$new_vid \leftarrow v; new_view \leftarrow V$
 send message (`flush`, vid , new_vid , i , (s_1, \dots, s_n)) to all servers in $view$
 block the application

upon receiving a message (`resend`, vid , \mathcal{M}) from P_j :

$resent \leftarrow resent \cup \mathcal{M}$

upon receiving (`flush`, vid , new_vid , j , $(s'_{j1}, \dots, s'_{jn})$) messages from all $P_j \in view \cap new_view$:

for $P_k \in view$ **do**
 $h_k \leftarrow \max_{\ell} \{s'_{k\ell}\}; l_k \leftarrow \min_{\ell} \{s'_{k\ell}\}$
 if $s_k = h_k$ **then**
 $\mathcal{M} \leftarrow \{(j', s', m') \in delivered \mid j' = k \text{ and } s' \in [l_k, h_k]\}$
 send message (`resend`, vid , \mathcal{M}) to all servers in $view \cap new_view$
 while $s_k < h_k$ **do**
 $s_k \leftarrow s_k + 1$
 wait for $(k, s_k, m) \in resent$ for some m
 v -deliver(m)
 output v -change(new_vid , new_view)
 $vid \leftarrow new_vid; view \leftarrow new_view$
 $new_vid \leftarrow 0; new_view \leftarrow \perp$
 $delivered \leftarrow \emptyset; resent \leftarrow \emptyset$
 unblock the application

If a server crashes during the view change, then the membership service eventually installs a new view and the protocol starts over with the next view.

¹As elsewhere, this notation means a message of the form (`send`, v , s' , m) with $v = vid$, since the global variable vid is already bound in this scope, and arbitrary s' and m , since s' and m are free variables.

If the group is stable, then the membership service will eventually install the same view at all group members. Hence, all members who transition together from some view to the new view compute the same *cut*, i.e., the set of maximal sequence numbers h_k for $P_k \in \text{view}$. The protocol then *v-delivers* all messages from P_k up to sequence number h_k ; since P_k may have crashed, the missing messages are recovered from the other members of *new_view* who must have remembered them.

With this mechanism, all servers *v-deliver* the same set of messages in *view* before installing *new_view*.

Because all $P_j \in \text{new_view}$ receive the same *v-change* event, they all send a `flush` message for *new_view*. Hence, P_i eventually receives such messages from all members of the new view that proceed to the new view. Likewise, the server who delivered the message with the maximal sequence number for P_k proceeds also to the new view and generates a `resend` message. Hence, all servers in *new_view* eventually install the new view.

If there are no further view changes, the set of delivered messages has to be garbage-collected eventually. For knowing when it is safe to forget a delivered message, a separate “stability mechanism” is usually employed. This is a protocol that periodically informs all servers about the messages delivered by all others.

Note that although applications relying virtually synchronous broadcast can be expressed asynchronously, the synchrony assumption is encapsulated in the membership service. Chockler et al. [CKV01] survey the specifications of various group communication systems. The view-synchronous broadcast algorithm above is a simplified version of the algorithm by Keidar et al. [KSMD02, KK02].

References

- [BJ87] K. P. Birman and T. A. Joseph, *Reliable communication in the presence of failures*, ACM Transactions on Computer Systems **5** (1987), no. 1, 47–76.
- [CKV01] G. V. Chockler, I. Keidar, and R. Vitenberg, *Group communication specifications: A comprehensive study*, ACM Computing Surveys **33** (2001), no. 4, 427–469.
- [IBM06] *Distribution & consistency services (DCS): Towards flexible levels of availability*, http://www.research.ibm.com/compsci/project_spotlight/distributed/dsc/, 2006.
- [KK02] I. Keidar and R. Khazan, *A virtually synchronous group multicast algorithm for WANs: Formal approach*, SIAM Journal on Computing **32** (2002), no. 1, 78–130.
- [KSMD02] I. Keidar, J. Sussman, K. Marzullo, and D. Dolev, *Moshe: A group membership service for WANs*, ACM Transactions on Computer Systems **20** (2002), no. 3, 191–238.
- [Pow96] D. Powell (Guest Ed.), *Group communication*, Communications of the ACM **39** (1996), no. 4, 50–97.