Dynamic Voting for Cluster Membership

Revision 0.2

Membership Service, OSDL Clustering Project

http://developer.osdl.org/dev/osdlclusters/

Primary Author(s): **Bob Spencer**

Key Contributors:

Revision History

Revision	Date	Author	Reason for Changes
0.1	February 20, 2004	BS	Initial Revision

Related Documents

Document Name	Revision
[1] Dynamic voting for consistent primary components. E. Yeger Lotem, I. Keidar, and D. Dolev.	
[2] Availability study of dynamic voting algorithms. K Ingols and I Keidar.	
[3] OSDL cluster architecture. Joe DiMartino, John Cherry, Daniel McNeil	
[4] Node discovery algorithm. Crystal Xiong	0.1
[5] CGL HA Clustering Glossary	

Glossary

For definitions regarding CGL clustering, see [5].

New terms not defined in CGL document:

Term	Definition
Quorum	(also referred to as primary quorum) The group of nodes that represent the cluster. Only nodes that are part of the quorum provide services to clients outside the cluster. When a node or network failure occur, there is temporarily no quorum. The voting algorithm defines how a new quorum is created using the remaining active nodes.

1. Introduction

To achieve high availability, a cluster may be used to provide continuous service to clients without disruption, even in the event of node failure within the cluster. A cluster uses membership to identify how the nodes are grouped together and identifies a primary group, or *quorum,* which provides services to those outside the cluster. When a node in the quorum fails or is removed from the cluster, It is imperative the cluster quickly create a new quorum and identify its membership. If successful, the cluster can effectively hide the failure from the clients it serves.

This document outlines how the dynamic voting algorithm and protocol described by Yeger Lotem et al. [1] (henceforward YKD) can be used in clusters for maintaining a quorum. YKD provides many features and advantages over other similar protocols as described below, namely excellent handling of dynamically changing membership with low probability of blocking and minimal communication rounds.

1.1 Protocol features

YKD achieves a balance between reliability, network traffic, and resiliency to blocking when failures occur during protocol execution. Its key features are:

1) Dynamic protocol well suited for unreliable networks or environments with dynamically changing number of nodes.
2) Requires minimal communication between nodes to form a quorum. Using local history storage, only two phase commit is required to reliably form a new quorum.
3) Guarantees no more than one quorum is formed after a node failure
4) Effectively handles failures that occur while the protocol is running (see [2]).
5) Requires a fixed minimum number of nodes to form a quorum to prevent the case where a majority of nodes cannot form a new quorum because of the potential existence of a past surviving quorum that contains a small set of nodes.

1.2 Assumptions

The following are assumed when the YKD protocol is started:

1) A quorum already exists before a failure. All members of the quorum have an accurate list of all other members.
2) Each node has a membership module (e.g. TIPC) that senses failures and recoveries of other nodes. “The protocol…is *correct* regardless of whether the membership mechanism is accurate or not. The *liveness* of the protocol (its ability to form new [quorums] when the network situation changes) depends on the accuracy and liveness of this membership mechanism.” [1].
3) It is possible from the membership message to derive a list of all visible nodes of our potential next quorum. This is necessary since the protocol will block while waiting for messages from these nodes. (Q6)

2. YKD protocol

The voting algorithm is initiated by all nodes when a membership message is received from the membership module indicating that a node within the quorum has left (failed or voluntarily exited). Membership messages indicating node recoveries or announcing new nodes do not require execution of the protocol. New nodes may join the quorum without running this protocol. (Q9)

2.1 Overview

Following a failure notification, the following steps are taken by all nodes:

1) Send node state information to all other nodes. **(Q2)** State information is:
   a. the last quorum in which this node had membership
   b. all ambiguous quorums attempted *since* this node was a member of a quorum. Note: the history of ambiguous quorums is erased when a node becomes a member of a formed quorum.
2) Check received state from all nodes and verify whether the new group of nodes can form the cluster's quorum. **(Q3)** In general (there are exceptions), to form a new quorum the following must be true:
   a. The new group must have the majority of the members of the last quorum.
   b. The new group must have the majority of the members all the ambiguous quorums attempted after the quorum.
3) If the criteria for forming a new quorum are met, attempt to form the quorum by taking the following steps:
   a. Record the new quorum as an ambiguous quorum **(Q5)**
   b. Send an attempt message to all nodes in the new group (now an ambiguous quorum)
4) If an attempt message is received from all other nodes in the new group, form the quorum:
   a. Update state with new quorum and remove ambiguous quorum information (as recorded in 3a).

Failures: Failures (additional nodes leaving the quorum) may occur during any stage of the protocol. In these cases the old protocol is immediately terminated and a new one is initiated (see Section 4, pg 5 of [1]).

Multiple attempts: A single run of the protocol could have multiple simultaneous quorum attempts pending.

Limit on Ambiguous Quorums: The number of ambiguous quorums that are attempted could be exponential but is limited to the number of nodes in the system (using local computation only). In practice, Ingols and Keider in [2] states that in 600,000 runs of the YKD protocol using 64 processes the most ambiguous quorums ever attempted was four (4) which occurred only twice.

2.2 Details

The section is derived from section 4 of [1]. It is broken down into two main sections: Step-by-step protocol description (2.2.2) and pseudo-code (2.2.3). The former is easier to read but the latter contains more depth.

2.2.1 Notation, Variables, and Structures

The items described in this section are used in both the following sections.

2.2.1.1 Notation

N = this node.

W = Membership of initial quorum. (a NodeList structure)

M = Members of new potential quorum after a failure (derived from membership message).

MIN_QUORUM_SIZE = The minimum number of nodes necessary to form a quorum.

2.2.1.2 Variables

These variables are maintained by all nodes

Variable	Type	Description
isPrimary	boolean	TRUE if N is part of cluster quorum
sessionNumber	long	Current session number. Incremented during YKD protocol.
lastPrimary	Quorum structure	Contains the SessionNumber and membership list of all nodes in last quorum in which N was a member.
ambiguousQuorumList	QuorumList structure	Quorum structures for pending (attempted) quorums that never succeeded.

2.2.1.3 Structures

Structure	Definition	Comment
Node	Sring nodeName void otherData	otherData might be network address or other useful data for this node
Quorum	NodeList memberList long sessionNumber	
NodeData	long sessionNumber Quorum lastPrimary QuorumList ambiguousQuorumList	Each node maintains this data. (See variables section)
NodeList	List of Node structures	
QuorumList	List of Quorum structures	
NodeDataList	List of NodeData structures	

2.2.2 Protocol Step-by-step

For those who prefer the pseudo-code, please jump to section 2.2.3. This section lists the steps that are presented in the pseudo-code.

1. Startup w/corresponding membership service

2. Initialize the members of N's NodeData structure:

Variable	Initial Value
isPrimary	Boolean flag. TRUE if part of quorum
sessionNumber	0
lastPrimary memberList sessionNumber	 W – if N is a member of W **(Q4)**, else NULL 0 – if N is a member of W, else -1
ambiguousQuorumList	Empty

3. Wait for membership messages

4. Receive node failure message
   A node that is a member of the quorum is no longer available (e.g. TIPC heartbeat has detected loss of the node). Based on the message, the new potential quorum membership can be derived. This new membership is referred to as M.

5. Execute the YKD protocol
   This will identify whether M can become the cluster quorum. If so the new quorum is formed. If a failure occurs during execution, the current run is aborted and the protocol is restarted.

   5.1. Send our node's NodeData to all members of M
   If error, exit protocol. If new failure membership message received, restart protocol.

   5.2. Receive NodeData from all members of M and construct a NodeDataList
   Wait until all members of M respond or another membership message arrives. If a failure occurs, restart the protocol

   5.3. Append our NodeData to the NodeDataList
   This is done for simplicity. In later computations we can assume that all the members of M (including N) are part of the NodeDataList.

   5.4. Compute MaxSession
   Find the largest SessionNumber in M

   5.5. Compute MaxPrimary
   Look at the "lastPrimary" value (last formed quorum) for all nodes in M and save the one that has the largest SessionNumber

   5.6. Compute MaxAmbiguousSessionList
   Get a list of all the ambiguous sessions that have formed since the MaxPrimary session calculated in the previous step. The SessionNumber identifies the order the ambiguous quorums were formed.

5.7. Compute *Sub_Quorum* (MaxPrimary, M)
(Decide if M can become the cluster quorum based on MaxPrimary)

Section 4.1 of [1] details how to verify whether the current membership M qualifies to become the new quorum. Using the variables computed in the previous 3 steps (MaxSession, MaxPrimary, and MaxAmbiguousSessionList), we can see if M meets the criteria.

The first step is to prove that *Sub_Quorum* (MaxPrimary, M) is TRUE.
Sub_Quorum (MaxPrimary, M) is TRUE iff
1) 5.7.1 is TRUE, and
2) 5.7.2 is TRUE or
5.7.3 is TRUE or
5.7.4 is TRUE

5.7.1. Compute $| M \cap W | \geq$ MIN_QUORUM_SIZE (See if the number of nodes in M that were part of W is at least MIN_QUORUM_SIZE)

5.7.2. Compute $| M \cap \text{MaxPrimary} | \geq | \text{MaxPrimary} | / 2$ (See if the number of nodes in M that were part of the latest quorum (MaxPrimary) is at least ½ the number of members of MaxPrimary.)

5.7.3. Compute $| M \cap \text{MaxPrimary} | == | \text{MaxPrimary} | / 2$ AND
$\exists p \in M \cap \text{MaxPrimary}$ s.t. $\forall q \in \text{MaxPrimary} - M$ $L(p) > L(q)$
(First line: If the number of nodes in M that were part of MaxPrimary, is *exactly* ½ the number of members of MaxPrimary
THEN (in order to break the tie)
Second line: Find the node whose name comes first in an alphabetical listing of all MaxPrimary's member's names. See if this node is in M.)

The second line broken down: for each node, p, that is a member of both M and MaxPrimary, compare p against all nodes, q, that are members of *only* MaxPrimary and see if p has a superior lexicographical order.

5.7.4. Compute $| M \cap W | > n -$ MIN_QUORUM_SIZE (See if the number of nodes M and W have in common is greater than the number of nodes only in W (n) minus MIN_QUORUM_SIZE)

5.8. ($\forall$Quorum $\in$ MaxAmbiguousSessionList),
Compute *Sub_Quorum* (Quorum.NodeList, M)
(Decide if M can become the new quorum based on the ambiguous quorums stored in MaxAmbiguousSessionList)

Re-execute all the steps described in section 5.7 for each quorum in MaxAmbiguousSessionList. If ALL of them return success, then we have succeeded and M can become the new cluster quorum.

5.9. Attempt to make M the cluster quorum

NodeData.sessionNumber = MaxSession + 1
Create a new Quorum: AttemptQuorum
AttemptQuorum.memberList = M
AttemptQuorum.sessionNumber = MaxSession + 1
add AttemptQuorum to NodeData.ambiguousQuorumList

5.10. Send *attempt* message to all members of M

If error, exit protocol. If new failure membership message received, restart protocol.

5.11. Receive *attempt* message from all members of M

Block until all members respond or another membership message arrives. If a failure occurs, restart the protocol

5.12. Success: Form Quorum

This step is just updating N's local variables. N assumes that all other nodes have received the attempt message and that the quorum is formed. If another node does not receive the attempt message then it will be handled as a failure notification and a new run of the protocol will be started.

NodeData.lastPrimary = newQuorum
NodeData.ambiguousQuorumList = 0
isPrimary = TRUE

2.2.3 Pseudo-code

```
declare myNodeData as NodeData           // data that node N shares with other nodes
declare M as NodeList                    // list of members of new potential quorum
declare W as NodeList                    // list of members of original quorum

function main ( )
{
    <startup>
    init ( )

    while <process not terminated>
    {
        <wait for membership message OR process termination request>

        if < membership message == node failure or loss >
        {
            declare failedNode as Node
            failedNode = <failed node from failure message>

            // start with last-known good membership list and remove failed node
            M = myNodeData.lastPrimary.memberList
            remove failedNode from M

            formPrimaryQuorum ( )
        }
        else
        {
            <handle other messages or requests>
        }
    }
}

function init ( )
{
    // assumption: N is member of initial quorum
    isPrimary = true                            //set to false if not member of W
    myNodeData.sessionNumber = 0                //always set to 0
    myNodeData.lastPrimary.memberList = W       //set to NULL if not member of W
    myNodeData.lastPrimary.sessionNumber = 0    //set to -1 if not member of W
    // myNodeData.ambiguousQuorumList is empty
}
```

```
61  function formPrimaryQuorum ( )
62  {
63      // send our node's data to all members of M
64      if ( sendSessionData ( M, myNodeData ) == false )
65          return false
66
67      // receive node data from all members of M
68      declare nodeDataList as NodeDataList
69      if ( receiveSessionData ( M, nodeDataList ) == false )
70          return false;
71
72      // To simplify future computations, append myNodeData to the NodeDataList just received
73      add myNodeData to nodeDataList
74
75      declare maxSession as long                  // The biggest session number of any node in M.
76      declare maxPrimary as Quorum                // Most recent quorum any node in M was part of
77      declare maxAmbSessionList as QuorumList     // List of ambiguous quorums attempted since maxPrimary
78
79      maxSession =          getMaxSession ( nodeDataList )
80      maxPrimary =          getMaxPrimary ( nodeDataList )
81      maxAmbSessionList =   getMaxAmbSessionList ( nodeDataList, maxPrimary.sessionNumber )
82
83      // first check with maxPrimary
84      if ( isQuorumAllowed ( maxPrimary, M ) == false
85          return false;
86
87      // now verify with all quorums in maxAmbSessionList
88      declare index as integer
89      for index from 0 to getCount (maxAmbSessionList ) by 1
90      {
91          if ( isQuorumAllowed ( maxAmbSessionList[index], M ) == false
92              return false;
93      }
94
95      // attempt to form quorum
96      myNodeData.sessionNumber = maxSession + 1
97      declare newQuorum as Quorum
98      newQuorum.memberList = M
99      newQuorum.sessionNumber = maxSession + 1
100     add newQuorum to myNodeData.ambiguousQuorumList
101
102     if ( sendAttempt ( M ) == false )
103         return false
104     if ( receiveAttempt ( M ) == false )
105         return false;
106
107     // form quorum
108     myNodeData.lastPrimary = newQuorum
109     empty myNodeData.ambiguousQuorumList
110     myNodeData.isPrimary = true
111
112     return true
113 }
114
115 function sendNodeData (NodeList memList, NodeData nodeData)
116 {
117     if < nodeData successfully sent to nodes in memList >
118         return true
119     else
120         return false
121 }
122
123 function receiveNodeData (NodeList memList, NodeDataList nodeDataList);
124 {
125     < wait until NodeData is received from all nodes in memList OR timeout OR process termination request>
126     if < node data successfully received from all nodes >
127         return true
128     else
129         return false
130 }
```

```
131
132     function getMaxSession ( NodeDataList nodeDataList )
133     {
134         // Find the largest SessionNumber in nodeDataList
135         declare maxSession as long
136         maxSession = -1
137
138         declare index as integer
139         for index from 0 to getCount ( nodeDataList ) by 1
140         {
141             if (maxSession < nodeDataList[index].sessionNumber)
142                 maxSession = nodeDataList[index].sessionNumber
143         }
144         return maxSession
145     }
146
147     function getMaxPrimary ( NodeDataList nodeDataList )
148     {
149         // Look at all the primary quorums (last formed) in M and save the one that has the largest SessionNumber
150         declare maxPrimary as Quorum
151         maxPrimary = NULL
152
153         declare index as integer
154         for index from 0 to getCount ( nodeDataList ) by 1
155         {
156             if ( maxPrimary == NULL OR
157                 maxPrimary.sessionNumber < nodeDataList[index].maxPrimary.sessionNumber )
158                  maxPrimary = nodeDataList[index].lastPrimary
159         }
160         return maxPrimary
161     }
162
163     function getMaxAmbSessionList ( NodeDataList nodeDataList, long maxPrimarySessionNumber )
164     {
165         // Get a list of all the ambiguous quorums that have formed since the last quorum.
166         // The input variable maxPrimarySessionNumber is used to identify session order.
167
168         declare maxAmbSessionList as QuorumList
169         declare index1 as integer
170         declare index2 as integer
171         declare nodeData as NodeData            // for readability
172         declare ambQuorum as Quorum             // for readability
173
174         for index1 from 0 to getCount ( nodeDataList ) by 1
175         {
176             nodeData = nodeDataList[index1]
177             for index2 from 0 to getCount ( nodeData.ambiguousQuorumList ) by 1
178             {
179                 ambQuorum = nodeData.ambiguousQuorumList[index2]
180                 if ( ambQuorum.sessionNumber > maxPrimarySessionNumber )
181                     add ambQuorum to maxAmbSessionList
182             }
183         }
184         return maxAmbSessionList
185     }
186
187     function isQuorumAllowed ( Quorum Q, NodeList M )
188     {
189         // Two boolean values are computed below: conditionA (min size) and conditionB (membership OR size)
190         // Sub_Quorum (MaxPrimary, M) = (conditionA AND conditionB)
191         declare conditionA as boolean
192         declare conditionB as boolean
193         conditionA = false
194         conditionB = false
195
196         declare memList_MW as NodeList                    // members of BOTH M and W
197         declare memList_MQ as NodeList                    // members of BOTH M and Q
198         memList_MW = getCommonMembers ( W, M )
199         memList_MQ = getCommonMembers ( Q.memberList, M )
200
```

```
201         conditionA = getCount ( memList_MW ) > MIN_QUORUM_SIZE
202         if ( conditionA )
203         {
204             if ( getCount (memList_MQ) > ( getCount ( Q.memberList ) / 2 ) )
205                 conditionB = true
206             else if ( getCount (memList_MQ) == ( getCount ( Q.memberList ) / 2 ) )
207             {
208                 // M contains exactly half of Q.memberList, need to check names of nodes to break tie
209                 declare firstMember as Node
210                 firstMember = getLexicographicalLeader ( Q.memberList )
211                 if ( isMember ( firstMember, M ) )
212                     conditionB = true
213             }
214             else if ( getCount (memList_MW ) > getCount (W) – MIN_QUORUM_SIZE )
215                 conditionB = true
216         }
217         return (conditionA AND conditionB)
218     }
219
220     function getCommonMembers ( NodeList memListA, NodeList memListB)
221     {
222         declare memListBoth as NodeList
223         declare index1 as integer
224         declare index2 as integer
225
226         for index1 from 0 to getCount ( memListA ) by 1
227         {
228             for index2 from 0 to getCount ( memListB ) by 1
229             {
230                 if ( memListA[index1] == memListB[index2] )
231                     add memListA[index1] to memListBoth;
232                     break
233             }
234         }
235         return memListBoth
236     }
237
238     function getLexicographicalLeader ( NodeList memList )
239     {
240         declare index as integer
241         declare firstMember as Node
242         firstMember = NULL
243
244         // find the node with the name at the top of an alphabetical listing
245         for index from 0 to getCount ( memList ) by 1
246         {
247             if ( firstMember == NULL OR
248                stringCompare (firstMember.nodeName, memList[index]) < 0 )
249                 firstMember = memList[index]
250         }
251         return firstMember
252     }
253
254     function sendAttempt ( NodeList memList )
255     {
256         if < attempt message successfully sent to nodes in memList >
257             return true
258         else
259             return false
260     }
261
262     function receiveAttempt ( NodeList memList )
263     {
264         < wait until attempt message is received from all nodes in memList OR timeout OR process termination request>
265         if < attempt message successfully received from all nodes >
266             return true
267         else
268             return false
269     }
```

2.3 Code structures

The following is a first-pass at the implementation of the structures identified in section 2.2.1.3

```
int isPrimary;                          /* 1 if this node is a member of the quorum, 0 if it is not. */

typedef struct Node_s {
        char *name;
        //otherdata x;
} Node;

typedef struct Quorum_s {
        NodeList *members;              /* List of names of nodes that are part of this quorum. */
        long sessionNumber;             /* Session number of this quorum, or -1 if members is empty */
} Quorum;

typedef struct NodeData_s {
        long sessionNumber;             /* The current session number.  Initialized to 0 and updated in
                                        the Attempt step (see 5.9 above) */
        Quorum *primaryQuorum;          /* Points to information about the last quorum this node was a
                                        member of.  Initialized to NULL if this node is not a member of
                                        the initial quorum */
        QuorumList *pendingQuorums;     /* List of ambiguous quorums attempted (see 5.9 above) since
                                        the last formed quorum this node was a member of.   Initialized
                                        to NULL */
} NodeData;

typedef struct NodeList_s {
        Node member;
         struct NodeList_s *next;
} NodeList;

typedef struct QuorumList_s {
        Quorum *q;
        struct QuorumList_s *next;
} QuorumList;

typedef struct NodeDataList_s {
        QuorumData *qData;
        struct NodeDataList_s *next;
} NodeDataList;
```

2.4 History Storage

2.5 Flow chart

2.6 Sequence Diagram

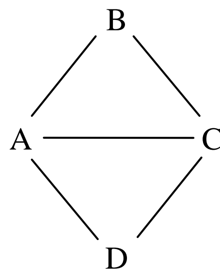
2.7 State Machine

3. Questions

Q1) If multiple services are provided by single system, then will multiple instances of TIPC heartbeat be running on a single system representing each service? If so, is that efficient? If not, how will the group of nodes know that a particular service has failed if the system is still successfully sending/receiving heartbeats?

A1) For membership, we are only trying to handle problems with the single system. 1 copy of TIPC runs for this node and tracks which other nodes we can communicate with. Service failure (node failure) is not handled by cluster membership, is has to be handle separately.

Q2) How and to whom does a node send its state after a failure? Is it multicast out? If not, how do nodes outside the current group get involved? For example, in the following diagram if {ABC} is the quorum and node B fails, A and C need to exchange state with D to form a new quorum {A,C,D}. Since D was not part of the original quorum how does D get involved?



Section 4 of [1] suggests that all nodes are known by all other nodes at the outset. So the answer could be that all nodes always send failure notifications to all known nodes, regardless of whether they are part of the current quorum. If that is the case, how does a new node become known to existing nodes (e.g. how to introduce a new system into the cluster)?

A2) The protocol should use multicast. In your example, when {ABC} was formed without D, A and C both know about D, but excluded it since it could not talked to B. When B dies, A and C should now know that they can let D join. (actually D would know too since A and C would have sent their connectivity info to D during the previous attempt at membership). D just has to wait for A and C to say, Hey, you can join now!

Q3) After sending out its initial state after a failure, how long should a node wait to receive state from other nodes? If it attempts to create a quorum each time it receives a new node's state then quorums will always initially be the minimum size.

A3) A node will know M when it gets the failure (from heartbeat). The thread executing the current algorithm will wait indefinitely (or until another failure message) for responses from all members of M.

Q4) How does the node get the initial membership list (and contact information) of the members of the initial quorum?

A4) A configuration file will be used to identify the initial members, minimum quorum size, and a flag whether to require all nodes specified in the config file to form the initial quorum or to form a quorum when the minimum quorum size is available. (Other config data tbd)

Q5) How is history stored?

A5) We only need history for this "node", so I was thinking of storing it on the root file system somewhere.

Q6) How do new nodes join the quorum?

4. Scenario Examples

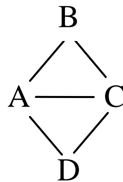
How is M determined? If M = potential membership of new quorum and we use multicast, then how do we know when we've received *all* responses from members? I've walked through the protocol with a few scenarios and still am trying to figure out how M is derived and how to make sure that all nodes attempting a quorum have the same M.

4.1 Example 1

Numbering below is from steps in section 2.2.2.

1. Startup w/corresponding membership service

Assume we start with the following setup. {A,B,C} have initial quorum.



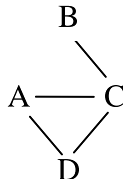
2. Initialize the members of N's NodeData structure:

A,B,C state:
isPrimary = 1;
sessionNumber = 1;
lastPrimary.memberList = {A,B,C};
lastPrimary.sessionNumber = 1;
ambSessionList = NULL;
D state:
isPrimary = 0;
sessionNumber = -1;
lastPrimary = NULL;
ambSessionList = NULL;

3. Wait for membership messages

4. Receive node failure message

Assume that the link between A and B is lost.



Who gets the notification (assume everyone)? Does the notification state that B is lost or just that some failure occurred within the primary quorum? (B isn't really lost from C's point of view) At this point, how do the nodes get M? If the membership service (e.g. heartbeat) provides M, then A&D have M={A,C,D}, B has M={B,C}, and C has M={A,B,C,D}.
OK, stopping here as this may not be a valid scenario...