The Voting Farm
A Distributed Class for Software Voting

Vincenzo De Florio
Universiteit Antwerpen
Department of Mathematics and Computer Science
MOSAIC research group
Middelheimlaan 1, B-2020 Antwerpen

(version revised on April 29, 2015)
1. VotingFarmTool.

This document describes a class of C functions implementing a distributed software voting mechanism for EPX [11, 12] or similar message passing multi-threaded environments. Such a tool may be used for example, to set up a restoring organ [9] i.e., an NMR (i.e., N-module redundant) system with N voters.

In order to describe the tool we start defining its basic building block, the voter.

A voter is defined as a software module connected to one user module and to a farm of fellow voters arranged into a clique.

![Diagram of a user module and its local voter.](node_users_voter.png)

**Figure 1.** A user module and its local voter.

By means of the functions in the class the user module is able:

- to create a static “picture” of the voting farm, needed for the set up of the clique;
- to instantiate the local voter;
- to send input or control messages to that voter.
Figure 2. The architecture of the voting farm: each user module connects to one voter and interacts only with it. In particular, the user module sends its local voter only one input value; the voter then broadcasts it across the farm; then it receives $N - 1$ messages from its fellows so to be able to perform the voting.

No interlocutor is needed other than the local voter. The other user modules are supposed to create coherent pictures and instances of voters on other nodes of the machine and to manage consistently the task of their local intermediary. All technicalities concerning the set up of the cliqué and the exchange of messages between the voters are completely transparent to the user module. More information about the voting farm may be found in [6, 7, 8].

In the following the basic functionalities of the VotingFarm class will be discussed, namely how to set up a “passive farm”, or a non-alive (in the sense of [4, 5]) topological representation of a yet-to-be-activated voting farm; how to initiate the voting farm; how to control the farm.

\begin{itemize}
\item Global Variables and \# include’s 3
\item Voting Farm Declaration 4
\item Voting Farm Definition 6
\item Voting Farm Description 7
\item Voting Farm Activation 11
\item Voting Farm Control 14
\item Voting Farm Read 28
\item Voting Farm Destruction 27
\item Voting Farm Error Function 51
\end{itemize}
(Voting Algorithms 29)
(The Voter Function 30)
2. Prologue: headers, global variables, etc.

#define VF_MAX_NTS 16 /* size of stacks \equiv max value for N */
#define VOTING_FARMS_MAX 64 /* max number of simultaneous active voting farms */
#define VF_MAX_INPUT_MSG 512 /* max size of an input message */
#define VF_MAX_MSGS 10 /* max size of the message buffer */
#define VF_EVENT_TIMEOUT 10 /* Select time-out is 10 seconds */
#define NO 0
#define YES 1
#define E_VF_OVERFLOW -1 /* error conditions */
#define E_VF_CANT_ALLOC -2
#define E_VF_UNDEFINED_VF -3
#define E_VF_WRONG_NODE -4
#define E_VF_GETGLOBID -5
#define E_VF_CANT_SPAWN -6 /* CreateThread error */
#define E_VF_CANT_CONNECT -9 /* ConnectLink error */
#define E_VF_RECVLINK -7 /* RecvLink error */
#define E_VF_BROADCAST -8 /* Duplicated input message */
#define E_VF_DELIVER -10 /* Invalid output LinkCB_t */
#define E_VF_BUSY_SLOT -11 /* Invalid input message - can't broadcast */
#define E_VF_WRONG_VFID -12 /* Can't deliver */
#define E_VF_WRONG_DISTANCE -13 /* Can't connect */
#define E_VF_INPUT_SIZE -19 /* Can't broadcast */
#define E_VF_INACTIVE -14 /* Can't connect */
#define E_VF_UNDEFINED_VF -15 /* Error conditions */
#define E_VF_NO_LVOTER -16 /* Error conditions */
#define E_VF_WRONG_MSG_NB -17 /* Error conditions */
#define E_VF_SENDLINK -18 /* Error conditions */
#define E_VF_INPUT_SIZE -19 /* Error conditions */
#define E_VF_UNDESCRIBED -20 /* Error conditions */
#define E_VF_INACTVE -21 /* Error conditions */
#define E_VF_UNKNOWN_SENDER -22 /* Error conditions */
#define E_VF_EVENT_TIMEOUT -23 /* Error conditions */
#define E_VF_SELECT -24 /* Error conditions */
#define E_VF_WRONG_ALGID -25 /* Error conditions */
#define E_VF_NULLPTR -26 /* Error conditions */
#define E_VF_TOO_MANY -27 /* Error conditions */
#define E_VF_ERROR_NB 28 /* Error conditions */
#define E_VF_MAX_FARMS 64 /* Error conditions */
#define VFA_EXACT_CONCENSUS 0 /* Error conditions */
#define VFA_MAJORITY 1 /* Error conditions */
#define VFA_MEDIAN 2 /* Error conditions */
#define VFA_PLURALITY 3 /* Error conditions */
#define VFA_WEIGHTED_AVG 4 /* Error conditions */
#define VFA_SIMPLE_MAJORITY 5 /* Error conditions */
#define VFA_SIMPLE_AVERAGE 6 /* Error conditions */
#define VF_SUCCESS 1 /* Error conditions */
#define VF_FAILURE 0 /* Error conditions */
#define VF_NB_ALGS 7 /* nb of algorithms plus one */
#define VFD_EPSILON 5·10⁻⁵ /* Error conditions */
#define VFP_INITIALISING 0 /* Error conditions */
#define VFP_CONNECTING 1 /* Error conditions */


```c
#define VFP_BROADCASTING 2
#define VFP_VOTING 3
#define VFP_WAITING 4
#define VFP_FAILED 5
#define VFP_QUITTING 6
```
3. Two global variables have been supplied for the user to query the error status of the application and the name of the function which experienced the error. Their definition constitutes the main part of the following section.

(Global Variables and #include’s 3) ≡
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <sys/root.h>
#include <sys/logerror.h>
#include <sys/link.h>
#include <sys/select.h>
#include <sys/time.h>
#include <sys/thread.h>
#ifdef SERVERNET
#include "server.h"
#endif /* SERVERNET */

int VF_error; /* global variable for storing the error condition */
static double ε = VFD_EPSILON;
static double ScalingFactor = 1.0;
int once = 1;
static int VF_RequestId(int, int, int);
typedef struct {
    unsigned char *item;
    int item_nr;
} cluster_t; /* A flag is attached to each object so that the object can be logically “deleted” from the list simply setting its status to NOT_PRESENT. Once the list is created, all its elements are labeled as PRESENT; as the execution goes by, elements are “logically” removed from the list changing their status to NOT_PRESENT. */
typedef unsigned char flag;
typedef struct {
    void *object;
    flag status;
} value_t; /* This part has been added in V1.5. It defines a set of (redefineable) symbolic constants representing upper limits for pre-allocated areas used exclusively in the static version of the tool. */
#ifndef VF_STATIC_MAX_INP_MSG
#define VF_STATIC_MAX_INP_MSG 64
#endif
#ifndef VF_STATIC_MAX_LINK_NB
#define VF_STATIC_MAX_LINK_NB 16
#endif
#ifndef VF_STATIC_MAX_VOTER_INPUTS
#define VF_STATIC_MAX_VOTER_INPUTS 20
#endif
#ifndef STATIC
#define AllocationClass static
LinkCB_t *st_links[VF_STATIC_MAX_LINK_NB];
Option_t *st_options[VF_STATIC_MAX_LINK_NB];
double st_VFA_sum; /* used in VFA algorithms */
double st_VFA_weight[VF_STATIC_MAX_INP_MSG];
double st_VFA_squaredist[VF_STATIC_MAX_INP_MSG * VF_STATIC_MAX_INP_MSG];
cluster_t *st_clusters[VF_STATIC_MAX_VOTER_INPUTS];
void *st_voter_inputs[VF_STATIC_MAX_VOTER_INPUTS];
char st_chars[VF_STATIC_MAX_VOTER_INPUTS];
value_t st_VFA_v[VF_STATIC_MAX_INP_MSG];
char st_voter_inputs_data[VF_STATIC_MAX_VOTER_INPUTS][VF_STATIC_MAX_INP_MSG];
#endif
#define AllocationClass
#endif

unsigned char st_VFA_vote[VF_STATIC_MAX_INP_MSG];

This code is used in section 1.
4. Voting Farm Declaration. The whole VotingFarm class is built upon type Voting Farm_t, which plays the same role as the type FILE in the standard class of C functions for file management: it offers the user a way to refer to some object from an abstract point of view, masking him/her from all unneeded information concerning its implementation. All a user needs to know is that, in order to use a voting farm, he/she has first to declare an object like follows: VotingFarm_t * vf;

The newly defined vf variable does not describe any valid voting farm yet; it is simply a pointer with no object attached to it, exactly the same way it goes for a FILE *fp variable which has not been fopen’d yet. For that a special function is supplied: VF_open, which is discussed in the next subsection.

Each user module which needs to use a voting farm should declare a VotingFarm_t * variable.

\[\text{Voting Farm Declaration 4}\] ≡

```c
typedef struct {
    int vf_id;
    int vf_node_stack[VF_MAX_NTS];
    int vf_ident_stack[VF_MAX_NTS];
    LinkCB_t * pipe[2];
    int N;
    int user_thread;
    int this_voter;
    double (*distance)(void *, void *);
    flag broadcast_done;
    flag inp_msg_got;
    flag destroy_requested;
}#ifdef SERVERNET
    RTC_Thread rtc;
#endif
VotingFarm_t;
```

```c
static int VF_voter(VotingFarm_t *);
int VF_add(VotingFarm_t *, int, int);
void VF_perror(void);
#endif
VotingFarm_t Table[VF_MAX_FARMS];
``` 

This code is used in section 1.

5. In the above, vf_id is a unique integer which identifies a voting context, vf_node_stack is a stack of node ids (a voter thread will be spawned on each of them), vf_ident_stack is a stack of thread id, N is a stack pointer (only one stack pointer is needed because the two stacks evolve in parallel), and user_thread is the thread id of the caller (or user) module. pipe is a couple of pointers to LinkCB_t, used to communicate between the user module and the local voter. this_voter is the entry of the current voter.
6. Voting Farm Definition. A voting farm variable \( vf \) can be defined by means of function \( VF\_open \) e.g., as follows:

(1) \( \text{VotingFarm}_t *vf; \)
(2) \( vf = VF\_open(5, distance); \)

After this statement has been executed, an object has been allocated, some initializations have occurred, and the address of the newly created object has been returned into \( vf \). The number the user supplies as the argument of \( VF\_open \) is an integer which univokely represents the current voting farm and in fact distinguishes it from all other voting farms which possibly will be used at the same times—we may call it a VotingFarm-id. \( distance \) is an arbitrary metric i.e., a function which gets two pointers to opaque objects, computes a “distance”, and returns this value as a positive real number.

Each user module which needs to assemble the same voting farm should actually execute a \( VF\_open \) statement with the same number as an argument—it is the user’s responsibility to do like this. Likewise, coherent behaviour requires that the same metric function is referenced as second parameter of \( VF\_open \).

\( VF\_open \) returns \( \Lambda \) in case of error; otherwise, it returns a pointer to a valid object.

\[
\langle \text{Voting Farm Definition 6} \rangle \equiv \\
\text{VotingFarm}_t *VF\_open(\text{int } \_vf\_id, \text{double } (*distance)(\text{void } *, \text{void } *))
\]

```c
#define STATIC
static int \_vf\_max\_farms;
#endif
AllocationClass
    \text{VotingFarm}_t *vf;
    static char *VFN = "VF\_open";
    if (\_vf\_id \leq 0) {
        LogError(EC\_ERROR, VFN,
                "Illegal VotingFarm\_Identifier (\%d) --- should be greater than 0.", \_vf\_id);
        VF\_error = E\_VF\_WRONG\_VFID;
        return \Lambda;
    }
#endif STATIC
if ((vf = (\text{VotingFarm}_t *) malloc(sizeof(\text{VotingFarm}_t))) \equiv \Lambda) {
    LogError(EC\_ERROR, VFN, "Memory\_Allocation\_Error.");
    VF\_error = E\_VF\_CANT\_ALLOC;
    return \Lambda;
}
#else /* if STATIC is defined, then we fetch the next entry from array ‘Table’ */
if (\_vf\_max\_farms < \text{VF\_MAX\_FARMS}) \_vf = &\text{\_Table}\[\_vf\_max\_farms++];
else {
    LogError(EC\_ERROR, VFN, "Too\_many\_farms.");
    VF\_error = E\_VF\_TOO\_MANY;
    return \Lambda;
}
#endif
if (distance \equiv \Lambda) {
    LogError(EC\_ERROR, VFN, "Invalid\_Metric\_Function\_Distance\_NULL.");
    VF\_error = E\_VF\_WRONG\_DISTANCE;
    return \Lambda;
}
\_vf\_\_N = 0; /* zero the stack pointer */
\_vf\_\_id = \_vf\_id; /* record the vf id */
```
\begin{verbatim}
vf-distance = distance;  /* record the function pointer */
vf-user_thread = vf-this_voter = −1;
LocalLink(vf-pipe);       /* Create a means for communicating with the local voter */
\textbf{return} vf;
\end{verbatim}

This code is used in section 1.
7. Voting Farm Description. Once a VotingFarm_t pointer has been created and once an object has been correctly defined and attached to that pointer, the user needs to describe the farm: how many voters are needed, where they should be placed, how to refer to each voter, and so on. This is accomplished by means of function VF\_add. If the voting farm consists of \( N \) voters, then the user shall call \( VF\_add \) \( N \) times; each call describes a voter by attaching a couple \((n, t)\) to it, where \( n \) is the node of the voter and \( t \) is its thread identifier. As an example, the following statements:

\[
\begin{align*}
(1) & \text{VotingFarm_t } \ast vf; \\
(2) & \text{vf } = \text{VF\_open}(5, \text{distance}); \\
(3) & \text{VF\_add(vf, 15, tid1);} \\
(4) & \text{VF\_add(vf, 21, tid2);} \\
(5) & \text{VF\_add(vf, 4, tid5);} \\
\end{align*}
\]

declare (line (1)), define (line (2)), and describe (lines (3)–(5)) voting farm number 5. A triple of voters has been proposed; voter 0, identified by the couple \((n, t) = (15, \text{tid1})\), voter 1, or couple \((21, \text{tid2})\), and voter 2, or couple \((4, \text{tid5})\).

Again it is the user’s responsibility to operate in a coherent, consistent way during these phases: in this case, he or she needs to declare the farm in exactly the same order, with exactly the same cardinality, with the same attributes on all nodes. Exactly one node-id has to be present and equal to the number of the current node.

So far no thread has been launched, and no distributed action has taken place—therefore we talk of “passive voting farms” for farms that have been only declared, defined, and described, but not activated yet. \( VF\_add \) returns a negative integer in case of error; otherwise, it returns zero.

\[
\langle \text{Voting Farm Description} \rangle \equiv
\]

```c
int VF\_add(VotingFarm_t \ast vf, int node, int identifier)
{
    static int this_node; /* set function name */
    static char \*VFN = "VF\_add";
    if (this_node \equiv 0) this_node = GET_ROOT()->ProcRoot->MyProcID;
    \langle Has vf been defined? \rangle
    \langle Is vf a valid object? \rangle
    vf->node_stack[vf-N] = node;
    vf->ident_stack[vf-N] = identifier;
    if (node \equiv this_node)
    {
        vf->this_voter < 0) vf->this_voter = vf-N; /* store the current value of the stack pointer */
        else {
            LogError(EC_ERROR, VFN,"There must be only one local voter.");
            return VF\_error = E_VF_TOO\_MANY\_LVOTERS;
        }
        vf-N++;
        \langle Check stacks growth \rangle
        return VF\_error = 0;
    }
}
```

This code is used in section 1.
8. Checks if `vf` holds a valid (non-Λ) address.

\[
\langle \text{Has } \text{vf } \text{been defined? } 8 \rangle \equiv
\]

\[
\text{if } (\text{vf } \equiv \Lambda) \{
\text{LogError(EC_ERROR, VFN, } "\text{Undefined VotingFarm_t Object.}"");
\text{LogError(EC_ERROR, VFN, } "\text{t(A_VF_open_is_probably_needed)}"");
\text{return } \text{VF_error} = \text{E_VF_UNDEFINED_VF};
\}
\]

This code is used in sections 7, 11, 24, and 30.

9. Checks if `vf` points to valid data.

\[
\langle \text{Is } \text{vf } \text{a valid object? } 9 \rangle \equiv
\]

\[
\text{if } (\text{vf } \sim \text{vf_id} < 0 \lor \text{vf } \sim \text{vf_node_stack} \equiv \Lambda \lor \text{vf } \sim \text{vf_ident_stack} \equiv \Lambda \lor \text{vf } \sim N < 0) \{
\text{LogError(EC_ERROR, VFN, } "\text{Corrupted or Invalid VotingFarm_t Object.}"");
\text{return } \text{VF_error} = \text{E_VF_INVALID_VF};
\}
\]

This code is used in sections 7 and 30.

10. Check if a stack overflow event has occurred.

\[
\langle \text{Check stacks growth } 10 \rangle \equiv
\]

\[
\text{if } (\text{vf } \sim N \geq \text{VF_MAX_NTS}) \{
\text{LogError(EC_ERROR, VFN, } "\text{Stack Overflow.}"");
\text{LogError(EC_ERROR, VFN, } "\text{t(Increase the value of VF_MAX_NTS; current value is } %d\text{).}""),
\text{VF_MAX_NTS});
\text{return } \text{VF_error} = \text{E_VF_OVERFLOW};
\}
\]

This code is used in section 7.
11. **Voting Farm Activation.** After having described a voting farm, next step is turning that passive description into a “living” (active) object: this is accomplished by means of function $VF\_run$ which simply spawns the local voter and connects to it. Any inconsistency like e.g., zero or two local voters are managed at this point and results in specific error messages. The one argument `VotingFarm_t *vf` is passed to the newly created thread.

$VF\_run$ returns a negative integer in case of error; otherwise, it returns zero.

```c
⟨Voting Farm Activation 11⟩ ≡
int VF_run(VotingFarm_t *vf)
{
    AllocationClass
    int MyProcID;
    AllocationClass GlobId_t GlobId;
    AllocationClass int Error;
    #ifdef SERVERNET
    extern LinkCB_t *link2server;
    #endif
    /* set function name */
    static char *VFN = "VF_run";
    ⟨Has vf been defined? 8⟩
    ⟨Has vf been described? 12⟩
    if (vf-this_voter < 0) {
        LogError(EC_ERROR, VFN, "No voter has been defined to be local.");
        return E_VF_NO_LVOTER;
    }
    MyProcId = GET_ROOT()-ProcRoot-MyProcID;  /* Get the Global ID structure. */
    if (GetGlobId(&GlobId, Λ) ≡ −1) {
        LogError(EC_ERROR, VFN, "Cannot get the global ID of the thread.");
        return E_VF_GETGLOBID;
    }
    /* for the time being, this field is treated as a flag which tells whether $VF\_run$ has been executed or not on voting farm vf. Its role will be different when FT_CreateThread will be used instead of CreateThread. */
    vf-user_thread = 0;  /* first create a separate thread for the voter function */
    #ifdef SERVERNET
    LogError(EC_MESS, VFN, "Creating thread 'VF_voter()' via RTC_CreateLThread()");
    vf-rtc = RTC_CreateLThread (link2server, vf-vf_ident_stack[vf-this_voter], DIR_USER_TYPE, Λ, 0, (RTC_ptr_t)VF_voter, vf, sizeof ( LinkCB_t * ) );
    LogError(EC_MESS, VFN, "RTC_CreateLThread() has been executed.");
    #else
    if (CreateThread(Λ, 0, (int(*)( )) VF_voter, &Error, vf) ≡ Λ) {
        LogError(EC_ERROR, VFN, "Cannot start a voter thread, error code %d.", Error);
        return VF_error = E_VF_CANT_SPAWN;
    }
    #endif  /* SERVERNET */
    return VF_error = 0;
}
```

This code is used in section 1.
12. This checks whether the farm has been described by means of at least one call to VF_add.

\[ \langle \text{Has vf been described? 12} \rangle \equiv \]
\[
\text{if } (vf\_N < 0) \{
\text{LogError(EC\_ERROR, VFN, } \text{"Voting farm \%d needs to be described."}, vf\_vf\_id); \\
\text{LogError(EC\_ERROR, VFN, } \text{"(You probably need to execute a VF\_add\_statement.)"}); \\
\text{return } \text{VF\_error = E\_VF\_UNDESCRIBED};
\}
\]
This code is used in sections 11 and 24.

13. This checks whether the farm has been activated by means of a previous call to function VF_run.

\[ \langle \text{Has vf been activated? 13} \rangle \equiv \]
\[
\text{if } (vf\_user\_thread < 0) \{
\text{LogError(EC\_ERROR, VFN, } \text{"Voting farm \%d needs to be activated."}, vf\_vf\_id); \\
\text{LogError(EC\_ERROR, VFN, } \text{"(You probably need to execute a VF\_run\_statement.)"}); \\
\text{return } \text{VF\_error = E\_VF\_INACTIVE};
\}
\]
This code is used in section 24.
14. Voting Farm Control. All interactions between the user module and the farm go through the $\text{VF}_\text{control}$ and $\text{VF}_\text{control\_list}$ functions and objects of $\text{VF\_msg\_t}$ type, aka messages.

\[
\langle \text{Voting Farm Control} \rangle \equiv \\
\langle \text{Type } \text{VF\_msg\_t} \rangle \\
\langle \text{Build a } \text{VF\_msg\_t} \text{ message} \rangle \\
\langle \text{Function } \text{VF\_control\_list} \rangle \\
\langle \text{Function } \text{VF\_control} \rangle \\
\langle \text{Function } \text{VF\_send} \rangle \\
\]

This code is used in section 1.

15. A message is an object which holds the information needed for a user module to request a service to a voter and for a voter to respond to a previous user’s request. Its definition is simple:

\[
\langle \text{Type } \text{VF\_msg\_t} \rangle \equiv \\
\text{typedef struct} \\
\{ \\
\text{int code;} \\
\text{void *msg;} \\
\text{int msglen;} \\
\} \text{VF\_msg\_t};
\]

This code is used in section 14.

16. The $\text{code}$ field specifies the nature of the message (see table (see [table1])) for a complete reference); depending on this, some data may be pointed to by the opaque pointer $\text{msg}$. In this case, $\text{msglen}$ represents the size of that data. This is an example of its usage:

\[
(1) \text{VF\_msg\_t message;} \\
(2) \text{message.code} = \text{VF\_INP\_MSG}; \\
(3) \text{message.msg} = \text{strdup(input);} \\
(4) \text{message.msglen} = 1 + \text{strlen(input)};
\]

Note that the voter thread assumes that the user module allocates new memory for each new $\text{message.msg}$ that is sent to it i.e., \textit{it won’t make a personal copy of the area}; on the contrary, it will simply store the pointer and use the pointed storage. Moreover, also deallocation of objects previously defined by the user module will be considered to be managed by this latter.

17. These functions hide the $\text{VF\_msg\_t}$ structure to the user.

\[
\langle \text{Build a } \text{VF\_msg\_t} \text{ message} \rangle \equiv \\
\langle \text{Input message setup} \rangle \\
\langle \text{Scaling factor message setup} \rangle \\
\langle \text{Message to choose the algorithm} \rangle \\
\]

This code is used in section 14.
18. Build up an `VF_msg_t` object holding a `VF_INP_MSG` message.

\[
\text{\langle Input message setup 18 \rangle} \equiv \\
VF_msg_t *VFO_Set_Input_Message(\text{void *} \text{obj}, \text{size_t} \text{ siz}) \\
\begin{array}{l}
\text{static VF_msg_t } m; \\
\text{if (obj } \equiv \Lambda) \{ \\
\text{VF_error } = \text{E_VF_NULLPTR}; \\
\text{return } \Lambda; \\
\} \\
m.\text{code } = \text{VF_INP_MSG}; \\
m.\text{msg } = \text{obj}; \\
m.\text{msglen } = \text{siz}; \\
\text{return } \&\text{m}; \\
\end{array}
\]

This code is used in section 17.

19. Build up an `VF_msg_t` object holding a `VF_SCALING` message.

\[
\text{\langle Scaling factor message setup 19 \rangle} \equiv \\
VF_msg_t *VFO_Set_Scaling_Factor(\text{double *} \text{sf}) \\
\begin{array}{l}
\text{static VF_msg_t } m; \\
m.\text{code } = \text{VF_SCALING_FACTOR}; \\
m.\text{msg } = \text{sf}; \\
m.\text{msglen } = \text{sizeof(double)}; \\
\text{return } \&\text{m}; \\
\end{array}
\]

This code is used in section 17.

20. Build up an `VF_msg_t` object holding the chosen algorithm.

\[
\text{\langle Message to choose the algorithm 20 \rangle} \equiv \\
VF_msg_t *VFO_Set_Algorithm(\text{int} \text{ algorithm}) \\
\begin{array}{l}
\text{static VF_msg_t } m; \\
m.\text{code } = \text{VF_SELECT_ALG}; \\
m.\text{msglen } = \text{algorithm}; \\
\text{return } \&\text{m}; \\
\end{array}
\]

This code is used in section 17.

21. Function `VF_control_list` accepts an array of messages that are transferred across the communication network as one buffer.

Function `VF_control_list` and `VF_control` return a negative integer in case of error; otherwise, they return zero.
22. List of possible message codes going: from the user module to its local voter (represented as $U \rightarrow L$), and vice-versa (represented as $L \rightarrow U$):

- **VF_INP_MSG**: a msglen-byte-long input message is stored at the address referenced by the opaque pointer $msg$ ($U \rightarrow L$).
- **VF_OUT_LCB**: the information pointed to by $msg$ is the link control block for connecting the local voter to the output module ($U \rightarrow L$); see Figure 3.
- **VF_SELECT_ALG**: $msg$ points to a code which identifies a particular majority voting algorithm out of the set of available algorithms ($U \rightarrow L$).
- **VF_DESTROY**: a signal meaning that the receiving voter should terminate itself after having freed all no more needed memory ($U \rightarrow L$).
- **VF_NOP**: No OPeration, something like an ImAlive signal. For the time being this event is not used.
- **VF_RESET**: on the arrival of this signal the status is set so to be able to perform a new voting session with the current farm ($U \rightarrow L$).
- **VF_REFUSED**: certain operations may be refused; for example, if one tries to execute a $VF\_close()$ before a broadcasting operation has been completed, then the voter returns this message to its user module ($L \rightarrow U$).
- **VF_QUIT**: before quitting, the voter generates a $VF\_QUIT$ event ($L \rightarrow U$).
- **VF_DONE**: after broadcasting, a $VF\_DONE$ event is raised ($L \rightarrow U$).
- **VF_EPSILON**: An $\epsilon$ threshold value needed by the formalized majority voting algorithm ($U \rightarrow L$).
- **VF_ERROR**: A generic error has occurred ($L \rightarrow U$).
- **VF_SCALING_FACTOR**: used in the Weighted Averaging Technique (see below) ($U \rightarrow L$).

\[
\text{Output Node}
\]

\[
\text{OutputLink} = \text{ConnectLink}(\ldots) \quad \text{SendLink}(\text{OutputLink}, \text{vote}, \text{sizeof(vote)})
\]

\[
\text{User Module} \quad \text{OutputLink} \quad \text{Voter}
\]

**Figure 3.** The user module connects to an output module (dashed curve) and then sends to its local voter a $VF\_OUT\_LCB$ message holding the relevant link control. As soon as the voter has performed its voting task, it sends its vote to the output module by means of that link control block.

#define VF_INP_MSG  100
#define VF_OUT_LCB  101
#define VF_SELECT_ALG  102
#define VF_DESTROY  103
#define VF_NOP  104
#define VF_RESET 105
#define VF_REFUSED 106
#define VF_QUIT 107
#define VF_DONE 108
#define VF_EPSILON 109
#define VF_ERROR 110
#define VF_SCALING_FACTOR 111

23. List of possible message codes going from the local voter to its fellows and vice-versa. They have the same meaning of those defined in the previous section e.g.,VF_V_INP_MSG is an input message coming from a fellow voter.

#define VF_V_INP_MSG 200
#define VF_V_DESTROY 203
#define VF_V_NOP 204
#define VF_V_RESET 205
#define VF_V_ERROR 210
This function sends one or more messages to the local voter of voting farm $vf$:

$$\text{VF\_control\_list}(VotingFarm* vf, VF\_msg_t* msg, int n)$$

```
AllocationClass
  int rv;  /* set function name */
static char *VFN = "VF\_control\_list";
#ifndef VFDEBUG
  LogError(EC_MESS, VFN, "within \_control\_list");
#endif  /* VFDEBUG */
  (Has $vf$ been defined? 8)
  (Has $vf$ been described? 12)
  (Has $vf$ been activated? 13)
#ifndef VFDEBUG
  LogError(EC_MESS, VFN, "checked: defined, described, and activated");
#endif  /* VFDEBUG */
if ($n < 1 \lor n > VF\_MAX\_MSGS$) { /* doubt: should it be an error or a warning? */
  LogError(EC\_ERROR, VFN, "Wrong message number (%d) should be between 1 and %d.
   $n$, VF\_MAX\_MSGS);
  return E\_VF\_WRONG\_MSG\_NB;
}
if ($vf\_pipe[0] \neq \Lambda$) {
#ifndef VFDEBUG
  LogError(EC\_MESS, VFN, "Next statement is SendLink(%x, %x, \_d), \_pipe[0], &msg[0],
   n * sizeof (msg[0]))");
#endif  /* VFDEBUG */
  rv = SendLink($vf\_pipe[0], &msg[0], n * sizeof (msg[0]));
  if ($rv \neq n * sizeof (msg[0])) {
    LogError(EC\_ERROR, VFN, "Cannot send the message to local voter.");
    return E\_VF\_SEND\_LINK;
  }
#endif VFDEBUG
  LogError(EC\_MESS, VFN, "exiting...");
#endif  /* VFDEBUG */
  return VF\_error = 0;
}
LogError(EC\_ERROR, VFN, "Corrupted or Invalid VotingFarm_t Object.");
return VF\_error = E\_VF\_INVALID\_VF;
```

This code is used in section 14.
25. To be described...

\( \langle \text{Function } \text{VF}_\text{send} \ 25 \rangle \equiv \)

```c
#define VF_MAXARGS 31

void VF_send(VotingFarm_t *vf, int argc, ...)
{
    AllocationClass
    va_list ap;
    AllocationClass
    VF_msg_t array[VF_MAXARGS];
    AllocationClass
    VF_msg_t *mp;
    AllocationClass
    int i;
    if (argc > VF_MAXARGS) argc = VF_MAXARGS;
    va_start(ap, argc);
    for (i = 0; i < argc; i++) {
        mp = va_arg(ap, VF_msg_t *);
        memcpy(array + i, mp, sizeof(VF_msg_t));
    }
    va_end(ap);
    VF_control_list(vf, array, argc);
}
```

This code is used in section 14.

26. Simply a shortcut.

\( \langle \text{Function } \text{VF}_\text{control} \ 26 \rangle \equiv \)

```c
int VF_control(VotingFarm_t *vf, VF_msg_t *msg)
{
    return VF_control_list(vf, msg, 1);
}
```

This code is used in section 14.

27. Voting Farm Destruction. This is another simple shortcut for sending the \text{VF}_\text{DESTROY} message to a voting farm.

\( \langle \text{Voting Farm Destruction } 27 \rangle \equiv \)

```c
int VF_close(VotingFarm_t *vf)
{
    AllocationClass
    VF_msg_t msg;
    AllocationClass
    int r;
    msg.code = VF_DESTROY;
    r = VF_control(vf, &msg);
    return r;
}
```

This code is used in section 1.
28. to be described.

\<Voting Farm Read 28\> ≡

\[VF._msg.t *VF._get(VotingFarm.t *vf)\]

\{\n    static VF._msg.t msg;
    AllocationClass
    int recv, n;
    AllocationClass Option_to[5];  /* used to be Option_to[2] */
    static char *VFN = "VF._get";
    #ifdef VFDEBUG
    LogError(EC_MESS, VFN, "\␣\vf==%x, \␣\vf->pipe==%x, \␣\vf->pipe[0]==%x, \␣\vf->pipe[1]==%x", vf, 
             vf->pipe, vf->pipe[0], vf->pipe[1]);
    #endif  /* VFDEBUG */
    o[0] = ReceiveOption(vf->pipe[0]);
    o[1] = TimeAfterOption(TimeNow() + VF_EVENT_TIMEOUT * CLOCK_TICK);
    switch ((n = SelectList(2, o))) {
        case 1: LogError(EC_ERROR, VFN, "Timeout condition reached.");
            LogError(EC_ERROR, VFN, "\t(Maybe VF_EVENT_TIMEOUT should be enlarged? Now it's %d seconds.)", 
                     VF_EVENT_TIMEOUT);
            msg.code = VF_ERROR;
            msg.msglen = VF_error = E_VF_EVENT_TIMEOUT;
            return &msg;
        case 0:
            if ((recv = RecvLink(vf->pipe[0], (void *)&msg, sizeof (msg))) <= 0) {
                LogError(EC_ERROR, VFN, "Can't recv a message from the local voter");
                LogError(EC_ERROR, VFN, "\t(return code is %d), recv\t", recv);
                msg.code = VF_ERROR;
                msg.msglen = VF_error = E_VF_RECVLINK;
                return &msg;
            }
            break;
        default: LogError(EC_ERROR, VFN, "Can't Select, retvalue is %d", n);
            msg.code = VF_ERROR;
            msg.msglen = VF_error = E_VF_SELECT;
            return &msg;
    }  /* end switch */
    #ifdef VFDEBUG
    LogError(EC_MESS, VFN, "Received a message (%d bytes) from the voter.", recv);
    #endif  /* VFDEBUG */
    return &msg;
\}

static int voter_sendcode(LinkCB.t * UserLink, int code)
\{\n    AllocationClass
    int rv;
    AllocationClass
    VF._msg.t msg;
    static char *VFN = "voter_sendcode";
    msg.code = code;
    }
rv = SendLink(UserLink, &msg, sizeof (msg));

if (rv != sizeof (msg)) {
  LogError(EC_ERROR, VFN, "Cannot send the message to the user module.");
  return E_VF_SENDLINK;
}
return rv;
}

static int voter_sendmsg(LinkCB_t *UserLink, VF_msg_t *msg)
{
  AllocationClass
  int rv;
  static char *VFN = "voter_sendmsg";  /* static function ⇒ VFN is not touched */
  rv = SendLink(UserLink, msg, sizeof (*msg));

  if (rv != sizeof (*msg)) {
    LogError(EC_ERROR, VFN, "Cannot send the message to the user module.");
    return E_VF_SENDLINK;
  }
  return rv;
}

This code is used in section 1.

29. Voting functions are stored into the DoVoting array.

typedef struct {
    unsigned char *vote;
    int outcome;
} vote_t;

typedef void (*voter_t)(VotingFarm_t *, void *[], int, vote_t *);

static voter_t DoVoting[VF_NB_ALGS] = {
    VFA_ExactConcensus,
    VFA_MajorityVoting,
    VFA_MedianVoting,
    VFA_PluralityVoting,
    VFA_WeightedAveraging,
    VFA_SimpleMajorityVoting,
    VFA_SimpleAverage,
};

This code is used in section 1.
30. The Voter Function. The function at the basis of the voting thread.

\[
\text{The Voter Function, VF}\]

\[
\text{VF} = \text{static int VF\_voter(VotingFarm\_t *vf)}
\]

\[
\{
\text{LinkCB\_t * UserLink, *OutputLink, **links;}
\text{if defined SERVERNET}
\text{LinkCB\_t * serverlink = ConnectServer();}
\text{endif} /* SERVERNET */
\text{Option\_t * options;}
\text{int this\_voter, this\_node, N, i, opt, input\_nr;}
\text{VF\_msg\_t msg[VF\_MAX\_MSGS];}
\text{int msgnum;}
\text{VF\_msg\_t done;}
\text{int Algorithm = VFA\_MAJORITY;}
\text{char buffer[VF\_MAX\_INPUT\_MSG + sizeof(int)];}
\text{int Error, recv;}
\text{int input\_length;}
\text{void **voter\_inputs;}
\text{unsigned char *vote = st\_VFA\_vote;}
\text{vote\_t rvote;}
\text{unsigned int t0, tn; /* set function name */}
\text{static char *VFN = "VF\_voter";}
\text{t0 = TimeNow();}
\text{if defined SERVERNET}
\text{Set\_Phase(serverlink, VFP\_INITIALISING);}
\text{endif} /* SERVERNET */
\text{input\_nr = input\_length = 0;}
\text{UserLink = OutputLink = \Lambda;}
\text{vf\_broadcast\_done = vf\_inp\_msg\_got = vf\_destroy\_requested = NO;}
\text{rvote.vote;}
\text{if (GET\_ROOT()\_Proc\_Root\_My\_Proc\_ID \neq this\_node) }
\text{Log\_Error(EC\_ERROR, VFN, "Corrupted or Invalid VotingFarm\_t Object.");}
\text{return VF\_error = E\_VF\_INVALID\_VF;}
\text{}}
\text{N = vf\_\_N; /* Connect to the local Agent */}
\text{if defined SERVERNET}
\text{Ask the Server to set up a connection to an Agent 61}
\text{endif} /* SERVERNET */
\text{/* Get the ( LinkCB\_t * ) for communicating with the user module */}
\text{UserLink = vf\_pipe[1];}
\text{Create a cliqué 32}
\text{Poll the user link, the farm links, and the server link 37}
\text{}}
\text{An example of metric function 31}

This code is used in section 1.
31. An example of metric function: a double-returning version of `strcmp`.

\[
\text{double dstrcmp(void *a, void *b)}
\{
\quad \text{return (double) strcmp(a, b);} \\
\}
\]

This code is used in section 30.

32. A cliqué (fully interconnected crossbar) is set up among the voters. This is done in two “flavours:” in the STATIC one we make use of predefined static data, otherwise we perform `malloc()`’s.

\[
\text{#ifndef STATIC}
\]
\[
\text{#else}
\]
\[
\text{#endif} /* STATIC */
\]

This code is used in section 30.

33. An array of Link Control Block Pointers is needed in order to realize the cliqué.

\[
\text{links = (LinkCB_t **) malloc (N * sizeof (LinkCB_t *)) ; options = (Option_t **) malloc((N + 1) * sizeof (Option_t));}
\]
\[
voter_inputs = (void **) calloc(N, sizeof(void *));
\]
\[
\text{if (links \equiv \Lambda \land \text{ options } \equiv \Lambda \land \text{ voter_inputs } \equiv \Lambda) } \\
\quad \text{return VF_error = E_VF_CANT_ALLOC;}
\]

This code is used in section 32.

34. This section is supplied in case the user is willing to use the static version of this tool. Rather than allocating memory, we link statically-allocated memory to the appropriate pointers and we initialize them—where needed.

\[
\text{links = st_links;}
\]
\[
\text{options = st_options;}
\]
\[
\text{voter_inputs = st_voter_inputs;}
\]
\[
\text{memset(voter_inputs, 0, VF_STATIC_MAX_INP_MSG);}
\]

This code is used in section 32.
“By the implementation of queues the calling order \{to \textit{ConnectLink}\} does not matter.” [11]. As a consequence, a simple \texttt{for} should suffice to create the farm.

\textit{N}+1 options are “received”: \textit{N}-1 for the cliqué set-up, 1 for connecting to the user module, 1 for connecting to the server; more precisely:

- \texttt{options[\textit{this\_voter}]} regards the user module;
- \texttt{options[\textit{N}]} regards the local server;
- the rest regards the fellow voters.

\langle \text{Connect to your \textit{N}-1 fellows} \rangle \equiv

\#ifdef SERVERNET
\texttt{Set\_Phase (serverlink, VFP\_CONNECTING)};
\#endif

\begin{verbatim}
for (i = 0; i < \textit{N}; i++) {
  if (i \neq \textit{this\_voter}) {
    \texttt{links[i] = ConnectLink(\textit{vf\_node\_stack[i]}, VF\_RequestId(\textit{vf\_id}, i, \textit{this\_voter}), \&Error)};
    \texttt{options[i] = ReceiveOption(\texttt{links[i]});}
    if (\texttt{links[i]} \equiv \Lambda) {
      \texttt{LogError(EC\_ERROR, VFN, "Cannot connect to voter \%d."}, i);
      \texttt{return V\_error = E\_VF\_CANT\_CONNECT;}
    }
  }
  else {
    \texttt{options[i] = ReceiveOption(UserLink);}
  }
}
\#ifdef SERVERNET
\texttt{options[\textit{N}] = ReceiveOption(serverlink)};
\#endif  /* SERVERNET */
\end{verbatim}

This code is used in section 32.

36. This function creates a new request id beginning from voting farm id \texttt{vfn} and voter id’s \textit{v} and \textit{w}.

\begin{verbatim}
\textbf{static int} VF\_RequestId (int \textit{vfn}, int \textit{v}, int \textit{w})
{
  \textit{AllocationClass}
  int \textit{a, b};
  \textbf{static int} \texttt{hundreds = VF\_MAX\_NTS * VF\_MAX\_NTS;}
  if (\texttt{\textit{v}} > \texttt{\textit{w}}) \texttt{\textit{a} = \textit{w}, \texttt{\textit{b} = \textit{v}};
else \texttt{\textit{a} = \texttt{\textit{v}}, \texttt{\textit{b} = \textit{w};}
  \texttt{return hundreds} \cdot (\texttt{\textit{vfn}} + 1) + \texttt{\textit{a} \cdot VF\_MAX\_NTS} + \texttt{\textit{b};}
}
\end{verbatim}
37. If control reaches this section it means that the cliqué has been successfully established between the voters. All future actions depend on the control and input messages of the user; therefore, the voter puts itself into an “endless” loop waiting for new messages to arrive and to be managed. This event-driven loop indeed resembles the one that may be found in any X11 client to manage interactions with the keyboard, the pointer, the display server, and so on.

\( \langle \text{Poll the user link, the farm links, and the server link} \rangle \equiv */ t0 = \text{TimeNow();} */ \)

\[
\begin{align*}
\text{while (1)} \quad &
\begin{cases}
\text{if SERVERNET} \\
\text{else}
\end{cases} \\
&
\begin{cases}
\text{A message from the user module} \\
\text{A message from the cliqué} \\
\text{A message from the server module}
\end{cases}
\end{align*}
\]

\#ifdef SERVERNET
\begin{align*}
&\text{opt} = \text{SelectList}(N + 1, \text{options}); \\
&\text{return } \text{VF_error} = E_{\text{VF_UNKNOWN_SENDER}};
\end{align*}

\#endif /* SERVERNET */
\begin{align*}
\text{if (input_nr} \equiv N \land \text{vf-broadcast_done} \equiv \text{YES} \land \text{once}) \{ \\
&\begin{cases}
\text{FILE } *f; \\
\text{char } \text{fname}[80]; \\
\text{tn} = \text{TimeNow()};
\end{cases}
\end{align*}

\#ifdef TIMESMTS
\begin{align*}
&\text{sprintf}(\text{fname}, \text{"overhead.\%d", this_voter}); \\
&\text{f} = \text{fopen}(\text{fname}, \text{"a+"}); \\
&\text{fprintf}(f, \text{"%lu,%lu,%lf\n"}, t0, tn, (\text{double})(tn - t0)/\text{CLOCK_TICK}); \\
&\text{fclose}(f);
\end{align*}

\#endif
\begin{align*}
&\text{once} = 0;
\end{align*}

This code is used in section 30.
38. If the received options is option "this_voter", then a message is coming from the user module.

\(<\text{A message from the user module } 38>\) ≡

\[
\text{if (opt } \equiv \text{ this_voter) \{ /* Receives the data and checks whether it is an integral multiple of the message size or not. In that latter case, an error is issued. */}
\]

\[
\text{if ((msgnum } = \text{ RecvLink(UserLink, msg, VF_MAX_MSGS } \times \text{ sizeof } (*msg)) \% \text{ sizeof } (*msg)) \{ \\
\phantom{\text{if (}} \text{LogError(EC_ERROR, VFN, "Can't RecvLink[A_message_from_the_user_module]");} \\
\phantom{\text{if (}} \text{LogError(EC_ERROR, VFN, "size_of_the_message_is } \%d, \text{ sizeof(*msg)}_i \%d, (1)_i \%d, (2)_i \%d",} \\
\phantom{\text{if (}} \text{msgnum, sizeof(*msg), msgnum } \% \text{ sizeof } (*msg));} \\
\phantom{\text{if (}} \text{return VF_error } = \text{E_VF_RECVLINK;} \\
\}
\]

\[
\text{msgnum } \neq \text{ sizeof (*msg);}
\]

\#ifdef VFDEBUG

\[
\text{LogError(EC_MESS, VFN, "<voter } \%d_>_ \text{Received a_msg_from_the_user_module", this_node);} \\
\text{LogError(EC_MESS, VFN, "(code } \%d)_i \text{msgnum } \%d_>_ \text{Starting User_msg_management\n",} \\
\text{msg[0].code, msgnum);} \\
\]

\#endif

\{
\phantom{\text{if (}} \text{int i;} \\
\phantom{\text{if (}} \text{(<User message management 39)>}
\}
\]

This code is used in section 37.
39. A message from the user has been received. Deal with that message. (Note that messages are buffered into the \texttt{msg[]} array.)

\begin{verbatim}
⟨ User message management ⟩ ≡
    for (i = 0; i < msgnum; i++) { void *memdup(void *,size_t);
  #ifdef VFDEBUG
      LogError(EC_MESS,VFN,"<voter%d> User_message_management: loop%d code=%d\n",
            this_voter,i,msg[i].code);
  #endif
    switch (msg[i].code) {
      case VF_INP_MSG:
        #ifdef VFDEBUG
          LogError(EC_MESS,VFN,"<voter%d> VF_INP_MSG received.\n",
                this_node);
        #endif
          voter_inputs[this_voter] = memdup(msg[i].msg, msg[i].msglen);
          input_length = msg[i].msglen;
          input_nr++;  /* t0 = TimeNow(); */
        #ifdef VFDEBUG
          printf("<voter%d> message (as a double) is %lf.\n",
                this_node,*((double *) msg[i].msg));
        #endif
        ⟨ Check for a complete message suite; if so, vote, and possibly deliver the outcome ⟩
        vf_vec msg got = YES;
        if (this_voter ≡ input_nr - 1) {
            int j;
            ⟨ Broadcast the Input Message ⟩
            vf_vec broadcast done = YES;
        }
        break;
      ⟨ case VF_DESTROY: 41 ⟩  /* break; unneeded, because the statement is unreachable */
      ⟨ case VF_OUT_LCB: 42 ⟩ break;
    }
  case VF_SELECT_ALG: Algorithm = msg[i].msglen;
    break;
  case VF_SCALING_FACTOR: ScalingFactor = *((double *) msg[i].msg);
    break;
  case VF_EPSILON: \( \epsilon = \ast((\text{double} *) \text{msg[i].msg}); \)
    break;
  case VF_RESET: input_nr = input_length = 0;
    OutputLink = \Lambda, rvote.outcome = *vote = '\textbackslash 0';
    vf_vec broadcast done = vf_vec msg got = vf_vec destroy requested = NO;
  #ifndef STATIC
    AllocationClass
    int i;
    for (i = 0; i < vf_N; i++) {
        free(voter_inputs[i]);
        voter_inputs[i] = \Lambda;
    }
  #endif
  break;
\end{verbatim}
case VF_NOP: /* for the time being, nothing */
    break;

default: printf("<voter_%d>:<voter_%d>\n", this_voter, msg[i].code);
    break; } /* end switch */
} /* end for */

This code is used in section 38.

40. The actual transmission of this voter’s input message to all other voters in the farm. The buffer is built up so to tie the code of the message and the message itself.

Broadcast the Input Message

(Broadcast the Input Message 40) ≡

*(int *) buffer = VF_V_INP_MSG;
memcpy(buffer + sizeof(int), voter_inputs[this_voter], input_length);

#define SERVERNET
Set_Phase(serverlink, VFP_BROADCASTING);
#endif

#ifndef ZEROPERM
for (j = 0; j < N; j++) {
    if (j != this_voter) {
        LogError(EC_MESS, VFN, "<voter_%d> broadcasting (j==%d)\n", this_voter, j);
    }
    if (input_length + sizeof(int) != SendLink(links[j], buffer, input_length + sizeof(int))) {
        LogError(EC_ERROR, VFN, "Cannot SendLink to voter_%d.", j);
        return VF_error = E_VF_SENDLINK;
    }
}
#endif

else
for (j = this_voter + 1; j < N; j++) {
    if (input_length + sizeof(int) != SendLink(links[j], buffer, input_length + sizeof(int))) {
        LogError(EC_ERROR, VFN, "Cannot SendLink to voter_%d.", j);
        return VF_error = E_VF_SENDLINK;
    }
}
#endif

for (j = 0; j < this_voter; j++) {
    if (input_length + sizeof(int) != SendLink(links[j], buffer, input_length + sizeof(int))) {
        LogError(EC_ERROR, VFN, "Cannot SendLink to voter_%d.", j);
        return VF_error = E_VF_SENDLINK;
    }
}
#endif

This code is used in sections 39 and 45.
41. Management of a user message of type $VF\_DESTROY$.

\[
\begin{align*}
\text{\langle case VF\_DESTROY: 41 \rangle} & \equiv \\
\text{\texttt{case VF\_DESTROY:} } & \text{ /* ¡Broadcast a $VF\_V\_DESTROY$ event?? */} \\
\text{\texttt{if (vf\_broadcast\_done = NO \land vf\_N \neq 1) \{}} & \\
\text{\texttt{\quad voter\_sendcode(UserLink, VF\_REFUSED);}} & \\
\text{\quad break;}} & \\
\text{\texttt{\} else \{}} & \\
\text{\quad \#ifdef SERVERNET} & \\
\text{\quad \quad Set\_Phase(serverlink, VFP\_QUITTING);} & \\
\text{\quad \#endif SERVERNET} & \\
\text{\quad \{ int myident = vf\_ident\_stack[this\_voter]; int error ;}} & \\
\text{\quad \#ifdef DoBreakServer} & \\
\text{\quad \quad if ( (error = BreakServer(serverlink)) \neq 0) LogError(EC\_ERROR, VFN,}} & \\
\text{\quad \quad \quad "BreakServer\_failed,,error:%%d", error );} & \\
\text{\quad \#endif /* DoBreakServer */} & \\
\text{\quad \#ifdef VFDEBUG} & \\
\text{\quad \quad LogError(EC\_MESS, VFN,"<voter\_%d>\_myident=%d, Bye.\n", myident, this\_voter);} & \\
\text{\quad \#endif} & \\
\text{\quad \#endif /* SERVERNET */} & \\
\text{\quad exit(0);} & \\
\text{\quad return VF\_error = 0; \}}
\end{align*}
\]

This code is used in section 39.

42. Management of a user message of type $VF\_OUT\_LCB$.

\[
\begin{align*}
\text{\langle case VF\_OUT\_LCB: 42 \rangle} & \equiv \\
\text{\texttt{case VF\_OUT\_LCB: OutputLink = ( LinkCB_t *) msg[i].msg;}} & \\
\text{\texttt{if (OutputLink \neq A) \{}} & \\
\text{\quad if (rvote\_outcome \equiv VF\_SUCCESS) \{}} & \\
\text{\quad \quad (Deliver the Outcome 50)} & \\
\text{\quad \} \}} & \\
\text{\texttt{\} else \{}} & \\
\text{\quad LogError(EC\_ERROR, VFN,"Invalid\_output\_link\_control\_block\_can’t\_deliver.");} & \\
\text{\quad /* In the event of a $VF\_DESTROY$ message, the local voter should inform its fellow via the}} & \\
\text{\quad \quad \quad \quad broadcasting of a $VF\_V\_DESTROY$ event. ¡Broadcast a $VF\_V\_DESTROY$ event?? */} & \\
\text{\quad \}}
\end{align*}
\]

This code is used in section 39.
If the received options is not option # $N$ nor option # this_voter, then a message is coming from a voter in the farm. Note that this time messages cannot be buffered—only one message at a time will be received. Moreover, messages come from a different memory space—for this reason, the structure of the message can’t be that of a `VF_msg_t` object. A different approach must be used: an integer representing the code message should be directly attached to an opaque area. The resulting buffer constitutes the message.

\[
\langle \text{A message from the cliquée} \rangle \equiv \\
\text{if } (\text{opt} \neq \text{this_voter} \land \text{opt} \neq N) \{ \\
\text{if } ((\text{recv} = \text{RecvLink}(\text{links}[\text{opt}], \text{buffer}, \text{VF_MAX_INPUT_MSG})) < 0) \{ \\
\text{LogError(EC_ERROR, VFN, }"\text{Can}^{''}\text{t RecvLink[}A\text{ message from the clique]"}; \\
\text{LogError(EC_ERROR, VFN, }"\text{Sender is voter }%d, \text{size of message is }%d."\}, \text{opt, recv}); \\
\text{return } \text{VF_error} = E_{\text{VF_RECVLINK}}; \\
\} \\
\text{msg[0].code} = \ast(\text{(int *) buffer}); \\
\text{msg[0].msg} = \text{buffer} + \text{sizeof(int)}; \\
\text{msg[0].msglen} = \text{recv} - \text{sizeof(int)}; \\
\langle \text{Cliquée message management 44} \rangle \\
\}
\]

This code is used in section 37.
44. A new message has come from a voter in the cliqué.

\{ Cliqué message management 44 \} ≡

\begin{verbatim}
switch (msg[0].code) {
  case VF_V_INP_MSG:
    ifdef VFDEBUG
    printf( "voter%d received the input message from voter%d\n", this_voter, opt, *((double *) msg[0].msg));
    #endif
    if (input_length == 0) input_length = msg[0].msglen;
    else {
      if (input_length != msg[0].msglen) {
        LogError(EC_ERROR, VFN, "Wrong input size");
        return VF_error = E_VF_INPUT_SIZE;
      }
    }
    #ifndef STATIC
    if ((voter_inputs[opt] = (void *) malloc(input_length)) == 0) {
      LogError(EC_ERROR, VFN, "Memory Allocation Error.");
      return VF_error = E_VF_CANT_ALLOC;
    }
    #else
    voter_inputs[opt] = (void *) &st_voter_inputs_data[opt][0];
    #endif
    memcpy(voter_inputs[opt], msg[0].msg, input_length);
    input_nr++;
        \{ Check for a complete message suite; if so, vote, and possibly deliver the outcome \}
        \{ Check if it’s your turn to broadcast; if so, do it, and take note of that \}
    break;
  case VF_V_DESTROY:     \* Is there a use for such a message? \*/ \*/ for the time being, no action */
    #ifdef VFDEBUG
    LogError(EC_MESS, VFN, "voter%d received a VF_V_DESTROY message from voter%d\n", this_voter, opt);
    #endif
    break;
  case VF_V_ERROR:      \* The voter notifies an error condition \*/ \*/ for the time being, no action */
    break;
  case VF_V_RESET:      \* Is there a use for such a message? \*/ \*/ for the time being, no action */
    break;
  case VF_V_NOP:        \* for the time being, no action */
    break;
} \*/ end switch */
\end{verbatim}

This code is used in section 43.
45. Broadcasting is performed in an ordered fashion so to prevent deadlocks—a voter is allowed to broadcast only when the following two conditions hold at once:
- it has not performed a broadcast before, and
- the vote-id (a number from 0 to $vf\cdot N - 1$) is less than or equal to the current number of input messages that have been received (user message included), minus one.

\[
\text{if (this\_voter} \equiv \text{input\_nr} - 1) \{
\text{int } j;
\text{Broadcast the Input Message 40)
}\]

This code is used in section 44.

46. If the received options is option # N, then a message is coming from the local server module.

\[
\text{if (opt} \equiv N) \{
\text{int } recv;
\text{char msg}[\text{FTB\_ELEMENT\_SIZE};]
\text{if ((recv = RecvLink(serverlink, (\text{void }*) msg, \text{FTB\_ELEMENT\_SIZE}) \leq 0))}
\text{LogError(EC\_ERROR, VFN, "couldn't receive a message from the server\n");}
\text{else LogError(EC\_DEBUG, VFN, "got server message of %d bytes.\n", recv);}
\text{Server message management 47)}
\]

This code is used in section 37.

47. So far, an empty section.

\[
\text{Server message management 47) \equiv } \text{/* should deal with message kept in msg[], size recv. */}
\]

This code is used in section 46.

48. Every time a new message has come and consequently input\_nr has been incremented, we must check if it’s time for voting and if so, after voting, we check if we have an output address.

\[
\text{if (input\_nr} \equiv N) \{
\text{(Perform Voting 49)}
\text{if (OutputLink} \neq A) \{
\text{(Deliver the Outcome 50)}
\} \text{/* input\_nr = 0; */}
\}
\]

This code is used in sections 39 and 44.
49. The actual voting algorithm is managed by a function whose address is kept in the \textit{DoVoting} array at entry no. \textit{Algorithm}.

\begin{verbatim}
⟨Perform Voting 49⟩ ≡
  if (Algorithm ≥ 0 ∧ Algorithm < VF_NB_ALGS) {
    #ifdef SERVERNET
      Set_Phase(serverlink, VFP_VOTING);
    #endif
    DoVoting[Algorithm](vf, (void **) voter_inputs, input_length, &rvote);
  }
  else {
    LogError(EC_ERROR, VFN, "Wrong Algorithm number: %d, not in [0,%d[", Algorithm, VF_NB_ALGS);
    return VF_error = E_VF_WRONG_ALGID;
  }
  #ifdef VFDEBUG
    if (rvote.outcome ≡ VF_SUCCESS) {
      LogError(EC_MESS, VFN, "<voter %d> is sending a VF_DONE msg to the user--vote==%lf. \n", this_voter, *(double *) rvote.vote);
      printf("vote==%lf\n", *(double *) rvote.vote);
    }
    else LogError(EC_MESS, VFN, "Vote is undefined. ");
  #endif
  done.code = VF_DONE; /* possibly Λ, which means: “no unique vote is available” */
  done.msglen = rvote.outcome;
  done.msg = rvote.vote;
  recv = voter_sendmsg(UserLink, &done);
  #ifdef VFDEBUG
    LogError(EC_MESS, VFN, "<voter %d> sent a VF_DONE msg (%d bytes) to the user. \n", this_voter, recv);
  #endif
\end{verbatim}

This code is used in section 48.
The vote is sent to the output module.

\[ \langle \text{Deliver the Outcome} \rangle \equiv \]
\[
\begin{cases}
\text{if } (\text{vote.outcome} \equiv \text{VF_SUCCESS}) \\
\text{if } (\text{input.length} \neq \text{SendLink(OutputLink, vote, input.length)}) \\
\quad \text{LogError(EC_ERROR, VF, "Cannot deliver the output.";)} \\
\quad \text{return VF_error} = \text{E_VF_DELIVER;}
\end{cases}
\]

\#ifdef SERVERNET
\quad \text{Set Phase(serverlink, VFP_WAITING;)}
\#endif

\else 
\quad \text{char c} = 0;
\quad \text{if } (\text{SendLink(OutputLink, &c, 1)} \neq 1) \\
\quad\quad \text{LogError(EC_ERROR, VF, "Cannot deliver the negative result.");}
\quad\quad \text{return VF_error} = \text{E_VF_DELIVER;}
\}
\#ifdef SERVERNET
\quad \text{Set Phase(serverlink, VFP_FAILED;)}
\#endif
\}
\]

This code is used in sections 42 and 48.
§51. **VF\_error**: the `perror` function of the VotingFarm class. Statically defined as a vector of strings, its entries can be addressed as "-e", where `e` is the error condition returned in **VF\_error**. The number of messages has been specified so to reduce the risk of inconsistencies.

\[
\text{\{Voting Farm Error Function 51\} = }
\begin{array}{l}
\text{static char *errors}[\text{VF\_ERROR\_NB}] = \{"no\_error", \text{ /* no error */} \\
\text{"An\_internal\_stack\_has\_reached\_its\_upper\_limit", \text{ /* E\_VF\_OVERFLOW */} \\
\text{"The\_system\_was\_not\_able\_to\_execute\_allocation", \text{ /* E\_VF\_CANT\_ALLOC */} \\
\text{"This\_operation\_requires\_a\_defined\_voting\_farm", \text{ /* E\_VF\_UNDEFINED\_VF */} \\
\text{"A\_wrong\_node\_has\_been\_specified", \text{ /* E\_VF\_WRONG\_NODE */} \\
\text{"The\_system\_was\_not\_able\_to\_get\_the\_global\_id", \text{ /* E\_VF\_GETGLOBID */} \\
\text{"The\_system\_was\_not\_able\_to\_execute\_CreateThread", \text{ /* E\_VF\_CANT\_SPAWN */} \\
\text{"The\_system\_was\_not\_able\_to\_execute\_ConnectLink", \text{ /* ConnectLink error */} \\
\text{"The\_system\_was\_not\_able\_to\_execute\_RecvLink", \text{ /* E\_VF\_RECVLINK */} \\
\text{"The\_system\_was\_not\_able\_to\_perform\_broadcasting", \text{ /* E\_VF\_BROADCAST */} \\
\text{"Invalid\_output\_message\_size\_of\_the\_input\_message", \text{ /* E\_VF\_DELIVER */} \\
\text{"Duplicated\_input\_message", \text{ /* E\_VF\_BUSY\_SLOT */} \\
\text{"Invalid\_voting\_farm\_id", \text{ /* E\_VF\_WRONG\_VFID */} \\
\text{"Invalid\_metric\_function\_pointer", \text{ /* E\_VF\_WRONG\_DISTANCE */} \\
\text{"Inconsistent\_voting\_farm\_object", \text{ /* E\_VF\_INVALID\_VF */} \\
\text{"No\_local\_voters---one\_voter\_has\_to\_be\_specified", \text{ /* E\_VF\_NO\_LVOTER */} \\
\text{"More\_than\_one\_local\_voter\_has\_been\_specified", \text{ /* E\_VF\_TOO\_MANY\_LVOTER */} \\
\text{"A\_wrong\_number\_of\_messages\_has\_been\_specified", \text{ /* E\_VF\_WHONG\_MSG\_NB */} \\
\text{"The\_system\_was\_not\_able\_to\_execute\_SendLink", \text{ /* E\_VF\_SENDLINK */} \\
\text{"Inconsistency\_in\_the\_size\_of\_the\_input\_message", \text{ /* E\_VF\_INPUT\_SIZE */} \\
\text{"This\_operation\_requires\_a\_described\_voting\_farm", \text{ /* E\_VF\_UNDESCRIBED */} \\
\text{"This\_operation\_requires\_an\_active\_voting\_farm", \text{ /* E\_VF\_INACTIVE */} \\
\text{"Inconsistent\_voter\_id\_in\_sender\_unknown", \text{ /* E\_VF\_UNKNOWN\_SENDER */} \\
\text{"Time-out\_reached\_during\_a\_Select\()", \text{ /* E\_VF\_EVENT\_TIMEOUT */} \\
\text{"A\_Select\()\_returned\_an\_index\_out\_of\_range", \text{ /* E\_VF\_SELECT */} \\
\text{"Algorithm\_Id\_out\_of\_range", \text{ /* E\_VF\_WRONG\_ALGID */} \\
\text{"NULL\_in\_a\_call-by-reference\_pointer", \text{ /* E\_VF\_NULLPTR */} \\
\text{"Maximum\_number\_of\_opened\_voting\_farms\_exceeded", \text{ /* E\_VF\_TOO\_MANY */} \\
\} \\
\text{\}}
\end{array}
\]

\text{void VF\_error( void) 
\
\{
\text{static char *VFN = "VF\_error"; 
\text{if (VF\_error) 
\{
\text{fprintf(stderr, "Error\_condition\_number\_d\_raised\_while\_in\_function\_\%s:\\"\%s\"\n", 
VF\_error, VFN, errors[\_VF\_error]); 
\text{fflush(stderr); 
\}}
\}}
\]

This code is used in section 1.
52. The functions stored in the DoVoting array are defined here.

\[ \langle \text{Voting Functions} \rangle \equiv \]
\[ \langle \text{Exact Consensus} \rangle \]
\[ \langle \text{Majority Voting} \rangle \]
\[ \langle \text{Median Voting} \rangle \]
\[ \langle \text{Plurality Voting} \rangle \]
\[ \langle \text{Weighted Averaging} \rangle \]
\[ \langle \text{Simple Majority Voting} \rangle \]
\[ \langle \text{Simple Average} \rangle \]

This code is used in section 29.

53. The simplest algorithm, apart from exact consensus—counts the agreement and returns the widest.

\[ \langle \text{Simple Majority Voting} \rangle \equiv \]
static void VFA_SimpleMajorityVoting(VotingFarm_t *vf, void *inp[], int len, vote_t *vote)
{
  int i, j;
  int n = vf->n;
  int v[VF_MAX_NTS];
  int threshold;
  threshold = n >> 1; /* n/2; */
  for (i = 0; i < n; i++) v[i] = 0;
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      if (i != j) {
        if (vf->distance(inp[i], inp[j]) < VFD_EPSILON) {
          v[i]++;
        }
      }
  for (i = 0; i < n; i++)
    if (v[i] >= threshold) {
      vote->outcome = VF_SUCCESS;
      memcpy(vote->vote, inp[i], len);
      return;
    }
  vote->outcome = VF_FAILURE;
}

This code is used in section 52.
54. Exact consensus means perfect, bitwise equality.

\[
\langle \text{Exact Consensus 54} \rangle \equiv
\]

```c
static void VFA_ExactConcensus(VotingFarm_t *vf, void *inp[], int len, vote_t *vote) {
    int i;
    int n = vf->N;
    if (inp[0] == \Lambda) {
        vote->outcome = VF_FAILURE;
        return;
    }
    for (i = 1; i < n; i++) {
        if (inp[i] == \Lambda) {
            vote->outcome = VF_FAILURE;
            return;
        }
        if (memcmp(inp[0], inp[i], len) != 0) {
            vote->outcome = VF_FAILURE;
            return;
        }
    }
    vote->outcome = VF_SUCCESS;
    memcpy(vote->vote, inp[0], len);
}
```

This code is used in section 52.
55. Generalized Voters. Several commonly used voting techniques have been generalized in [10] to “arbitrary N-version systems with arbitrary output types using a metric space framework”, including:

- formalized majority voter (\texttt{VFA\_MAJORITY}; cf. [10, §2.1, pp.445–446]),
- generalized median voter (\texttt{VFA\_MEDIAN}; cf. [10, §2.2, p.447]),
- formalized plurality voter (\texttt{VFA\_PLURALITY}; cf. [10, §2.3, pp.447–448]), and the
- weighted averaging technique (\texttt{VFA\_WEIGHTED\_AVG}; cf. [10, §2.4, p.448]).

All these techniques are based on the concept of “metric space” which is now recalled:

A metric space is a couple \((X, d)\), where \(X\) is the output space of the voting threads and \(d\) is a real value function defined on \(X \times X\) which is able in some way to “compare” two objects belonging to \(X\); more precisely, \(d\) behaves as a “distance” measure of any two objects in \(X\). More formally, \(\forall (x, y, z) \in X^3\) the following properties hold:

1. \(d(x, y) \geq 0\) (distances are positive numbers or zeroes);
2. \(d(x, y) = 0 \Rightarrow x = y\) (different points have positive distances);
3. \(d(x, y) = d(y, x)\) (distances obey the reflexive property);
4. \(d(x, z) \leq d(x, y) + d(y, z)\) (two consecutive segments are greater than the segment that straightly connects their loose ends, unless the three points lie on the same straight line);

then \(d\) is called a “metric”.

In other words, a metric is a function which is able to compare any two input objects and is able to numerically express a degree of “closeness” between them. [10] shows how four different voting algorithms can be executed starting from such a function. It is the user responsibility to supply a valid metric function on the call to \texttt{VF\_open}: that function shall get two pointers to opaque objects, compute a “distance”, and return that value as a positive real number.

These functions take advantage of the \textit{Stack} class which has been used in order to mimic the list operations in the algorithms in [10].

\begin{verbatim}
 Majority Voting 55
 static void VFA\_MajorityVoting(VotingFarm\_t *vf, void *inp[], int len, vote\_t *vote)
 {
     int i;
     int n = vf->N;
     int v;
     cluster\_t *c;
     #ifndef STATIC
     c = calloc(n, sizeof(cluster\_t));
     #else
     c = st\_clusters;
     memset(c, 0, n * sizeof(cluster\_t));
     #endif
     // Create a partition of blocks which are maximal with respect to the metric property 59
     // v is set by <Create a partition...> to the cardinality of the partition */
     for (i = 0; i < v; i++) {
         if (c[i].item\_nr > n/2) {
             vote\_outcome = VF\_SUCCESS;
             memcpy(vote\_vote, c[i].item, len);
             return;
         }
     }
     vote\_outcome = VF\_FAILURE;
 }
\end{verbatim}

This code is used in section 52.
“Generalized Median Voter”, §2.2 of [10, p.447]. See also the “mid-value select” technique in [9, p.60]

#define PRESENT 1
#define NOT_PRESENT 0

#define MedianVoting

static void VFA_MedianVoting(VotingFarm_t *vf, void *inp[], int len, vote_t *vote)
{
    void *inputs[VF_MAX_NTS];
    value_t *v;
    int i, j;
    int ri, rj;
    double max, dist;
    int n, card;

    #ifndef STATIC
        static char *VFN="VFA/MedianVoting";
    #endif
    #ifdef STATIC
        v = st_VFA_v;
    #else
        v = (value_t *) malloc(len * sizeof(value_t));
        if (v == Λ)
        {
            LogError(EC_ERROR, VFN, "Memory Allocation Error.");
            VF_error = E_VF_CANT_ALLOC;
            return;
        }
    #endif

    for (n = vf->N, i = 0; i < n; i++)
    {
        v[i].object = inp[i];
        v[i].status = PRESENT;
    }

    ri = rj = 0;

    do 
    { 
        for (card = i = 0, max = -1.0; i < n; i++)
        { 
            if (v[i].status == PRESENT) 
            { 
                inputs[card++] = v[i].object;
                for (j = i + 1; j < n; j++)
                { 
                    if (v[j].status == PRESENT)
                    { 
                        if ((dist = (vf->distance)(v[i].object, v[j].object)) >= max)
                        { 
                            max = dist;
                            ri = i;
                            rj = j;
                        }
                    }
                }
            }
        }
        if (max != -1.0)
        { 
            v[ri].status = v[rj].status = NOT_PRESENT;
        }
    } while (card > 2);

    vote->outcome = VF_SUCCESS;
    memcpy(vote->vote, inputs[0], len);
}

This code is used in section 52.
57. “Formalized Plurality Voter”, §2.3 of [10, p.447].

\[
\text{Informalized Plurality Voter,}\text{ }\text{§2.3 of [10, p.447].}
\]

\[
\langle \text{Plurality Voting 57}\rangle \equiv
\]

\[
\text{static void } VFA\text{-PluralityVoting(VotingFarm_t } *vf, \text{ void } *\text{inp[]}, \text{int len,.vote_t } *\text{vote})
\]

\[
\}
\]

\[
\text{int } i, j;
\]

\[
\text{int } n = vf->N;
\]

\[
\text{int } v;
\]

\[
\text{int max};
\]

\[
\text{cluster_t } *c;
\]

\[
\#ifndef \text{STATIC}
\]

\[
c = \text{calloc}(n, \text{sizeof(cluster_t)});
\]

\[
\#else
\]

\[
c = st\text{-clusters};
\]

\[
\text{memset}(c, 0, n * \text{sizeof(cluster_t)});
\]

\[
\#endif
\]

\[
\text{if } (c \equiv \Lambda) \text{ LogError(EC_MESS, "Plurality", } \text{"c is NULL");}
\]

\[
\langle \text{Create a partition of blocks which are maximal with respect to the metric property 59}\rangle
\]

\[
\#ifdef \text{VFDEBUG}
\]

\[
\text{LogError(EC_MESS, "Plurality", } \text{"Partition has been created.");}
\]

\[
\#endif
\]

\[
\text{if } (max > 1) \}
\]

\[
\text{vote-outcome } = \text{VF\_SUCCESS}; \text{ } /* \text{memcpy(vote-}i\text{,vote, c[i].item, len}); */
\]

\[
\text{memcpy(vote-vote, c[j].item, len});
\]

\[
\}
\]

\[
\text{else } vote\text{-outcome } = \text{VF\_FAILURE;}
\]

\[
\#ifndef \text{STATIC}
\]

\[
\text{free}(c);
\]

\[
\#endif
\]

\[
\}
\]

This code is used in section 52.
“Weighted Averaging Technique”, §2.4 of [10, p.448]. The ScalingFactor variable is used for computing a set of “weights”, defined as follows: given \( n \) values, \( x_1, x_2, \ldots, x_n \), then

\[
\forall i \in \{1, 2, \ldots, n\} : w_i = \left[1 + \prod_{j=1, j\neq i}^{n} d^2(x_j, x_i) \right]^{-1}
\]

where \( a \) is equal to ScalingFactor and \( d \) is the metric. Considered \( S = \sum_{i=1}^{n} w_i \), the voted value is computed as \( x = \left( \frac{\sum_{i=1}^{n} w_i x_i}{S} \right) \). The voted value is computed as \( \frac{\sum_{i=1}^{n} w_i x_i}{S} \).

\[
\langle \text{Weighted Averaging} \rangle \equiv
\]

```c
static void VFA_WeightedAveraging(VotingFarm_t *vf, void *inp[], int len, vote_t *vote)
{
    int i, j;
    int n = vf->N;
    double *sum;
    double *weight, wsum;
    double *squaredist;
    double partial, f;
    int r, c;
    static char *VFN = "WeightedAveraging";
    
    #ifdef STATIC
        sum = &st_VFA_sum;
        weight = &st_VFA_weight;
        squaredist = &st_VFA_squaredist;
    #else
        sum = malloc(sizeof(double));
        weight = (double *) malloc(n * sizeof(double));
        squaredist = (double *) malloc(n * n * sizeof(double));
    #endif /* STATIC */

    if (sum == NULL || weight == NULL || squaredist == NULL) {
        LogError(EC_ERROR, VFN, "Memory Allocation Error.");
        VF_error = E_VF_CANT_ALLOC;
        return;
    }

    if (ScalingFactor == 0) {
        LogError(EC_MESS, VFN, "Illegal scaling factor. Set to 1");
        ScalingFactor = 1.0;
    } /* compute the distances */

    for (i = 0; i < n; i++)
        for (j = 0; j < i; j++)
            f = (vf->distance)(inp[i], inp[j]);
            squaredist[i * n + j] = f * f;

    for (wsum = 0.0, i = 0; i < n; i++)
        partial = 1.0;
    for (j = 0; j < n && j != i; j++)
        if (i < j)
            r = j, c = i;
        else
            r = i, c = j;
        partial *= squaredist[r * n + c];

    partial /= (ScalingFactor * ScalingFactor);
    wsum += weight[i] = 1.0 / (1.0 + partial);
```
\begin{verbatim}
for (*sum = 0.0, i = 0; i < n; i++) {
    *sum += (*(double *) inp[i]) * weight[i];
}
if (wsum != 0) {
    *sum /= wsum;
    vote->outcome = VF_SUCCESS;
    memcpy(vote->vote, sum, len);
} else vote->outcome = VF_FAILURE;
#endif
#endif
\end{verbatim}

This code is used in section 52.
VF GENERALIZED VOTERS

59. The input values are partitioned into a set of blocks, \( V_1, V_2, \ldots, V_n \), such that for each \( i \) block \( V_i \) is maximal with respect to the property that

\[
\forall (x, y) \in V_i \times V_i : d(x, y) \leq \epsilon,
\]

where \( d \) is the metric.

In order to reproduce as much as possible the algorithmic formalism of [10] we decided

- to mimic their Lisp-like statements with stacks, and
- to use `goto` statements.

In this way actions (1)–(6) in [10, p. 445–446] can be (more or less) mapped into the statements corresponding to labels `one` to `six` that follow.

\[ \langle \text{Create a partition of blocks which are maximal with respect to the metric property 59} \rangle \equiv \]

\[
\begin{align*}
\text{char} & \quad \text{vt; } \\
\text{char} \ast & \quad \text{del; } \\
\text{void} \ast & \quad \text{item; } \\
\text{int} & \quad i, j, \text{item\_nr; } \\
\text{vt} & = \text{GET\_ROOT}(\; )-\text{ProcRoot}-\text{MyProcID}; \\
\#ifndef & \quad \text{STATIC} \\
\text{del} & = \text{calloc}(n, 1); \quad /* \text{alloc + set all of them to NO} */ \\
\#else & \\
\text{del} & = \text{st\_chars; } \\
\text{memset} & (\text{del}, 0, n); \\
\#endif & \\
\text{for} & (v = i = 0; i < n; \; i++) \{ \\
& \text{if} (\text{del}[i]) \text{ continue; } \\
& \text{item} = \text{inp}[i]; \\
& \text{del}[i] = \text{YES}; \\
& c[v] . \text{item} = \text{item; } \\
& \text{for} (\text{item\_nr} = 1, j = i + 1; j < n; \; j++) \{ \\
& \text{if} (-\text{del}[j] \land \text{vf-distance(item, inp}[j]) < \epsilon) \{ \\
& \text{del}[j] = \text{YES}; \\
& \text{item\_nr}++; \\
& \} \\
& c[v] . \text{item\_nr} = \text{item\_nr}; \\
& v++; \\
& \} \\
\#ifndef & \quad \text{STATIC} \\
\text{free} & (\text{del}); \\
\#endif & \\
\end{align*}
\]

This code is used in sections 55 and 57.
60. Simple Averaging may be useful e.g., to “melt together” \( n \) sample values of a same pixel of an image. Of course it requires the objects are numbers. For the time being, the computation is performed in double precision floating point arithmetics. A problem of this technique is that it assumes the samples are not faulty, in the sense that they do not differ too much from each other: an enormously different addendum would cause the average to differ as well from the “correct” values. The weighted averaging technique is a partial solution to this.

\[
\text{Simple Average} = \frac{\sum \text{values}}{n}
\]

```c
static void VFA_SimpleAverage(VotingFarm_t *vf, void *inp[], int len, vote_t *vote)
{
    int i;
    int n = vf->N;
    double *sum;
    static char *VFN = "SimpleAverage";
    #ifdef STATIC
        sum = &st_VFA_sum;
    #else
        sum = malloc(sizeof(double));
        if (sum == NULL) {
            LogError(EC_ERROR, VFN, "Memory Allocation Error.");
            VF_error = E_VF_CANT_ALLOC;
            return;
        }
    #endif

    for (*sum = 0.0, i = 0; i < n; i++) {
        *sum += (*double *) inp[i];
    }

    if (n == 0) {
        LogError(EC_ERROR, VFN, "Inconsistency--farm cardinality should be zero.");
        VF_error = E_VF_INVALID_VF;
        return;
    }

    *sum /= n;
    vote->outcome = VF_SUCCESS;
    memcpy(vote->vote, sum, len);
    #ifndef STATIC
        free(sum);
    #endif
}
```

This code is used in section 52.

61. This means:
- send a message to the server telling it “connect me to the Agent (thread id 1)”
- do a `ConnectLink` with the Agent
- send a set up message to the Server so that it propagates that message to the Agent.

\[\text{Ask the Server to set up a connection to an Agent} 61 \equiv \quad /* \text{yet to be implemented} */\]

This code is used in section 30.
62. **Closings.** This document and source code describes the actual implementation of the Voting Farm Tool as it appears in the EFTOS [1, 2] Basic Functionality Set Library. It has been crafted by means of the CWEB system of structured documentation [3].
63. Index. Here is a list of the identifiers used, and where they appear. Underlined entries indicate the place of definition. Error messages are also shown.

- a: 31, 36
- algorithm: 20
- AllocationClass: 3, 6, 11, 24, 25, 27, 28, 36, 39
- op: 25
- argc: 25
- array: 25
- b: 31, 36
- BreakServer: 41
- broadcast_done: 4, 30, 37, 39, 41, 45
- buffer: 30, 40, 43
- c: 50, 55, 57, 58
- calloc: 33, 55, 57, 59
- card: 56
- CLOCK_TICK: 28, 37
- cluster_t: 3, 55, 57
- code: 15, 16, 18, 19, 20, 27, 38, 39, 43, 44, 49
- ConnectLink: 2, 35, 51, 61
- ConnectServer: 30
- CreateThread: 2, 11
- del: 59
- destroy_requested: 4, 30, 39
- DIR_USER_TYPE: 11
- dist: 56
- distance: 4, 6, 7, 53, 56, 58, 59
- DoBreakServer: 41
- done: 30, 49
- DoVoting: 29, 49, 52
- dstcmp: 31
- E_VF_BROADCAST: 2, 51
- E_VF_BUSY_SLOT: 2, 51
- E_VF_CANT_ALLOC: 2, 6, 33, 44, 51, 56, 58, 60
- E_VF_CANT_CONNECT: 2, 35
- E_VF_DELIVER: 2, 50, 51
- E_VF_EVENT_TIMEOUT: 2, 28, 51
- E_VF_GETGLOBID: 2, 11, 51
- E_VF_INACTIVE: 2, 13, 51
- E_VF_INPUT_SIZE: 2, 44, 51
- E_VF_INVALID_VF: 2, 9, 24, 30, 51, 60
- E_VF_NO_LVOTER: 2, 11, 51
- E_VF_NULLPTR: 2, 18, 51
- E_VF_OVERFLOW: 2, 10, 51
- E_VF_RECVLINK: 2, 28, 38, 43, 51
- E_VF_SELECT: 2, 28, 51
- E_VF_SENDLINK: 2, 24, 28, 40, 51
- E_VF_TOO_MANY: 2, 6, 51
- E_VF_TOO_MANY_LVOTER: 51
- E_VF_TOO_MANY_LVOTERS: 2, 7
- E_VF_UNDEFINED_VF: 2, 8, 51
- E_VF_UNDESCRIPTED: 2, 12, 51
- E_VF_UNKNOWN_SENDER: 2, 37, 51
- E_VF_WRONG_ALGID: 2, 49, 51
- E_VF_WRONG_DISTANCE: 2, 6, 51
- E_VF_WRONG_MSG_NB: 2, 24, 51
- E_VF_WRONG_NODE: 2, 51
- E_VF_WRONG_VFID: 2, 6, 51
- EC_DEBUG: 46
- EC_ERROR: 6, 7, 8, 9, 10, 11, 12, 13, 24, 28, 30, 33, 35, 37, 38, 40, 41, 42, 43, 44, 46, 49, 50, 56, 58, 60
- EC_MESS: 11, 24, 28, 39, 40, 41, 44, 48, 49, 57, 58
- e: 3, 39, 59
- Error: 11, 30, 35
- errors: 51
- exit: 41
- f: 37, 58
- fclose: 37
- fflush: 51
- flag: 3, 4
- fopen: 4, 37
- fp: 4
- fprintf: 37, 51
- free: 39, 57, 58, 59, 60
- FT_Create_Thread: 11
- FTB_ELEMENT_SIZE: 46
- GET_ROOT: 7, 11, 30, 59
- GetGlobId: 11
- GlobId: 11
- GlobId_t: 11
- hundreds: 36
- i: 25, 30, 38, 39, 53, 54, 55, 56, 57, 58, 59, 60
- identifier: 7
- inp: 53, 54, 55, 56, 57, 58, 59, 60
- inp_msggot: 4, 30, 39, 45
- input: 16
- input_length: 30, 39, 40, 44, 49, 50
- input_nr: 30, 37, 39, 44, 45, 48
- inputs: 56
- item: 2, 55, 57, 59
- item_nr: 3, 55, 57, 59
- j: 39, 45, 53, 56, 57, 58, 59
- len: 4, 53, 54, 55, 56, 57, 58, 60
- LinkCB_t: 2, 3, 4, 5, 28, 30, 33, 42
- links: 30, 33, 34, 35, 40, 43
- link2server: 11
- LocalLink: 6.
LogError:  6, 7, 8, 9, 10, 11, 12, 13, 24, 28, 30, 33, 35, 37, 38, 39, 40, 41, 42, 43, 44, 46, 48, 49, 50, 56, 57, 58, 60.
m: 18, 19, 20.
malloc:  4, 6, 32, 33, 46, 58, 60.
max: 56, 57.
memcmp: 54.
memcpy:  4, 25, 40, 44, 53, 54, 55, 56, 57, 58, 60.
memset: 34, 39.
message:  16.
msg: 15, 16, 18, 19, 20.
malloc:  4, 6, 32, 33, 44, 56, 58, 60.
max: 56, 57.
memcmp: 54.
memcpy:  4, 25, 40, 44, 53, 54, 55, 56, 57, 58, 60.
memset: 34, 39.
message:  16.
msg: 15, 16, 18, 19, 20, 22, 24, 28, 30, 35, 37, 38, 39, 42, 43, 44, 46, 48, 49, 50, 56, 57, 58, 60.
msglen: 15, 16, 18, 19, 20, 22, 24, 28, 30, 35, 37, 38, 39, 42, 43, 44, 46, 49.
msgnum: 30, 38, 39.
myident: 41.
MyProcID: 11.
MyProcId: 11.
N: 4, 30.
v: 24, 28, 30, 38, 39, 42, 43, 44, 46, 49.
NO: 2, 30, 39, 41, 45.
node: 7.
NOT_PRESENT: 3, 50.
obj: 18.
object: 3, 56.
one: 3, 37.
one: 59.
opt: 30, 37, 38, 43, 44, 46.
Option_t: 3, 28, 30, 33.
options: 30, 33, 34, 35, 37.
outcome: 29, 39, 42, 49, 50, 53, 54, 55, 56, 57, 58, 60.
OutputLink:  30, 39, 42, 48, 50.
p: 4.
partial: 58.
error: 51.
pipe: 4, 5, 6, 24, 28, 30.
PRESENT: 3, 56.
printf: 39, 44, 49.
ProcRoot: 7, 11, 30, 59.
q: 4.
r: 27, 58.
ReceiveOption: 28, 35.
recv: 28, 30, 43, 46, 49.
RecvLink: 2, 28, 38, 43, 46.
ri: 56.
rj: 56.
rtc: 4, 11.
RTCCreateLThread: 11.
RTCptr_t: 11.
RTCThread_t: 4.
rv: 24, 28.
vvote: 30, 39, 42, 49, 50.
ScalingFactor: 3, 39, 58.
Select: 2.
SelectList: 28, 37.
SendLink: 2, 24, 28, 40, 50.
serverlink: 30, 35, 40, 41, 46, 49, 50.
SERVERNET: 3, 4, 11, 30, 35, 37, 40, 41, 49, 50.
SetPhase: 30, 35, 40, 41, 49, 50.
sv: 19.
six: 59.
siz: 18.
sprintf: 37.
squaredist: 58.
sl_chars: 3, 59.
sl_clusters: 3, 55, 57.
sl_links: 3, 34.
sl_options: 3, 34.
sl_VFA_squaredist: 3, 58.
sl_VFA_sum: 3, 58, 60.
sl_VFA_v: 3, 56.
sl_VFA_vote: 3, 30.
sl_VFA_weight: 3, 58.
sl_voter_inputs: 3, 34.
sl_voter_inputs_data: 3, 44.
Stack: 55.
STATIC: 3, 4, 6, 32, 39, 44, 55, 58, 59, 60.
status: 3, 56.
stderr: 51.
strcmp: 31.
strdup: 16.
strlen: 16.
sum: 58, 60.
Table: 4, 6.
this_node: 7, 30, 38, 39.
this_voter: 4, 5, 6, 7, 11, 30, 35, 37, 38, 39, 40, 41, 43, 44, 45, 48, 49.
threshold: 53.
tid1: 7.
tid2: 7.
tid5: 7.
TimeAfterOption: 28.
TimeNow: 28, 30, 37.
TIMESTATS: 37.	in: 30, 37.
t0: 30, 37.
user_thread: 4, 5, 6, 11, 13.
UserLink: 28, 30, 35, 38, 41, 49.
v: 36, 53, 55, 56, 57.
va_arg: 25.
va_end: 25.
va_start: 25.
A message from the cliqué 43 Used in section 37.
A message from the server module 46 Used in section 37.
A message from the user module 38 Used in section 37.
Allocate an array of LinkCB_t pointers 33 Used in section 32.
An example of metric function 31 Used in section 30.
Ask the Server to set up a connection to an Agent 61 Used in section 30.
Broadcast the Input Message 40 Used in sections 39 and 45.
Build a VF_msg_t message 17 Used in section 14.
Check for a complete message suite; if so, vote, and possibly deliver the outcome 48 Used in sections 39 and 44.
Check if it’s your turn to broadcast; if so, do it, and take note of that 45 Used in section 44.
Check stacks growth 10 Used in section 7.
Cliqué message management 44 Used in section 43.
Connect to your N-1 fellows 35 Used in section 32.
Create a cliqué 32 Used in section 30.
Create a partition of blocks which are maximal with respect to the metric property 59 Used in sections 55 and 57.
Deliver the Outcome 50 Used in sections 42 and 48.
Exact Concensus 54 Used in section 52.
Function VF_controlList 24 Used in section 14.
Function VF_control 26 Used in section 14.
Function VF_send 25 Used in section 14.
Global Variables and #include’s 3 Used in section 1.
Has vf been activated? 13 Used in section 24.
Has vf been defined? 8 Used in sections 7, 11, 24, and 30.
Has vf been described? 12 Used in sections 11 and 24.
Initialize an array of LinkCB_t pointers 34 Used in section 32.
Input message setup 18 Used in section 17.
Is vf a valid object? 9 Used in sections 7 and 30.
Majority Voting 55 Used in section 52.
Median Voting 56 Used in section 52.
Message to choose the algorithm 20 Used in section 17.
Perform Voting 49 Used in section 48.
Plurality Voting 57 Used in section 52.
Poll the user link, the farm links, and the server link 37 Used in section 30.
Scaling factor message setup 19 Used in section 17.
Server message management 47 Used in section 46.
Simple Average 60 Used in section 52.
Simple Majority Voting 53 Used in section 52.
The Voter Function 30 Used in section 1.
Type VF_msg_t 15 Used in section 14.
User message management 39 Used in section 38.
Voting Algorithms 29 Used in section 1.
Voting Farm Activation 11 Used in section 1.
Voting Farm Control 14 Used in section 1.
Voting Farm Declaration 4 Used in section 1.
Voting Farm Definition 6 Used in section 1.
Voting Farm Description 7 Used in section 1.
Voting Farm Destruction 27 Used in section 1.
Voting Farm Error Function 51 Used in section 1.
Voting Farm Read 28 Used in section 1.
Voting Functions 52 Used in section 29.
⟨Weighted Averaging 58⟩ Used in section 52.
⟨case VF_DESTROY: 41⟩ Used in section 39.
⟨case VF_OUT_LCB: 42⟩ Used in section 39.
References

[1] EFTOS K.U.Leuven: The EFTOS Reference Guide and Cookbook. (EFTOS Deliverable 2.4.2, March 1997)
[2] Deconinck, G., and De Florio, V., and Lauwereins, R., and Varvarigou, T: EFTOS: A software framework for more dependable embedded HPC applications, accepted for presentation at the European Conf. in Parallel Processing (Euro-Par '97).
[3] Knuth, D.E.: Literate Programming (Center for the Study of the Language and Information, Leland Standard Junior University, 1992)
[4] Carriero, N., and Gelernter, D.: How to write parallel programs: a guide to the perplexed. ACM Comp. Surv. 21 (1989): 323-357.
[5] Carriero, N., and Gelernter, D.: LINDA in context. Comm. ACM 32 (1989): 444-458.
[6] De Florio, V., Deconinck, G., Lauwereins, R.: The EFTOS Voting Farm: a Software Tool for Fault Masking in Message Passing Parallel Environments. In Proc. of the 24th Euromicro Conference (Euromicro '98), Workshop on Dependable Computing Systems, Västerås, Sweden, August 1998. IEEE.
[7] De Florio, V., Deconinck, G., Lauwereins, R.: Software Tool Combining Fault Masking with User-defined Recovery Strategies. IEEE Proceedings – Software 145(6), 1998. IEEE.
[8] De Florio, V.: A Fault-Tolerance Linguistic Structure for Distributed Applications. Doctoral dissertation, Dept. of Electrical Engineering, University of Leuven, October 2000. ISBN 90-5682-266-7.
[9] Johnson, B.W.: Design and analysis of fault-tolerant digital systems. (Addison-Wesley, New York, 1989)
[10] Loreczak, P.R., and Caglayan, A.K., and Eckhardt, D.E.: A Theoretical Investigation of Generalized Voters. Proc. of the 19th Int.l Symp. on Fault Tolerant Computing, 1989: 444-451.
[11] Anonymous. Manual Pages of EPX 1.9.2. (Parsytec GmbH, Aachen, 1996)
[12] Anonymous. Embedded Parix Programmer’s Guide. In Parsytec CC Series Hardware Documentation. (Parsytec GmbH, Aachen, 1996)