Distributed Termination Detection by Counting Agent *

N. O. Garanina, E. V. Bodin

A.P. Ershov Institute of Informatics Systems,
Lavrent'ev av., 6, Novosibirsk 630090, Russia,
{garanina,bodin}@iis.nsk.su

Abstract. The detection of termination of a distributed computation is an important problem in distributed systems. A distributed computation is said to terminate when all its processes are passive and there is no unprocessed message in the communication channels. We suggest a new algorithm for the distributed termination detection (DTD) problem. Our algorithm exploits a special agent that accumulates knowledge about the activities of basic system processes, and can decide about system termination. This approach combines the benefits of both message-counting and credit/recovery DTD-algorithms. We prove correctness of this algorithm and introduce some significant modifications.

1 Introduction

The problem of detecting the termination of a distributed computation is well studied. The termination state of a distributed system is defined as the state in which there are no unprocessed messages in the system and all processes are waiting. Several distributed termination detection algorithms have been devised which differ from each other by various parameters, such as distributed network topology, communication channel types, fault tolerance, and more. Paper [8] provides a representative taxonomy of DTD-algorithms. We use that classification to describe some properties of termination detection algorithm.

Our proposed algorithm is similar to the message-counting algorithms, as e.g. [7, 9, 10], and to credit/recovery algorithms, in particular [11, 12]. In these message-counting algorithms, a special agent scans other processes for information about messages sent and received by the processes. In our case, an agent uses control messages sent by the basic processes to collect information about their current and potential activities. To some extent, our DTD-algorithm is inverse to the credit/recovery algorithms, in that all processes credit a single debtor –viz., the collector process– by sending it positive values representing their activity, and successively withdraw funds from it by sending negative values representing their termination. The key advantages of our algorithm is not to

* The research has been supported by Siberian Branch of Russian Academy of Science (Integration Grant n.15/10 "Mathematical and Methodological Aspects of Intellectual Information Systems").

require a traversal of the process network as in message-counting algorithms, nor to know the number of processes and manipulate real numbers as in traditional credit/recovery algorithms. Every active processes divides its credit for processes in communication, then passive processes return their part of the credit.

A simplified version of the protocol we present in this paper has first appeared in [3] in the context of termination detection for the multi-agent algorithm for ontology population based on the semantic analysis of unstructured data. In that work, basic information and rule agents perform a semantic analysis of input data, and the controller agent determines the moment when such basic agents cannot proceed processing any further.

The rest of the paper is organized as follows. Section 2 describes the base algorithm for the termination detection problem. Then Section 3 proves the basic properties of the algorithm, including its correctness. Section 4 presents some immediate modifications and generalisation of the basic algorithm. Finally, we conclude in Section 5 with a discussion of future research.

2 The Base Activity Balance Algorithm

For the base *Activity Balance algorithm* (*AB-algorithm*), we define a distributed system **AB** as a set of n *basic processes* p_i ($i \in [1..n]$) and *the controller* process C: $\mathbf{AB} = \{p_1, \ldots, p_n, C\}$. Basic processes are connected by reliable communication channels for messages. Such channels can be unidirectional or duplex. Besides every basic process is connected with the controller by a duplex channel. Messages are transmitted instantly and stored in channels until they are read. We assume no shared memory and clock between processes.

The basic processes $p_1, \ldots, p_n$ execute *the basic computation*. The messages they exchange in the basic computation are called *basic messages*. We consider a process *active* if it is processing its basic computation. Otherwise, the process is *passive*. Initially, each process in the system is either active or passive. Without loss of generality we assume that every basic process is active at the beginning. Only an active basic process may send a basic message to another basic process. A passive process can only become active if it receives a basic message. We say that a distributed system *terminates* if and only if every basic process is passive and every basic process' communication channel is empty.

In addition to its basic computation, each basic process executes additional actions used for determining system termination. These actions have no effect on the basic computation. In our base AB-algorithm, the extra actions involve only sending to the controller agent *control messages* relating their activity status. In general, control messages may contain any information. They are sent and received by both active and passive processes, and do not affect their active or passive status.

We only consider terminating basic processes, i.e., processes that do not run forever. This is equivalent to assert that the actions of every basic process between passive periods decreases some function with values in a well-founded set. Without loss of generality, we can select such function to depend the number

of basic messages over time. For example, it can be the sum of unread messages in the system at any given moment in time. We assume that the each process' basic computation is organized in three successive blocks: reading messages from the input (one at the time), processing messages, and sending messages (several simultaneously). The system's computation is kickstarted by *start messages* sent between basic processes. For simplicity we assume that each basic process can send start messages. We also assume that $M_p(t)$, the number of basic messages sent by the basic process p at its local time t, is a strictly decreasing function.

Let us now describe the protocols followed by the basic processes and by the controller. In the pseudo-code which follows, `get_mess(`*set*`)` denotes the function that removes an element from *set* and returns it. Also, we use C to indicate the controller. Variable `status` will be used to record the activity status of a basic process, viz., `active` or `passive`, integer `t` to mark of time of sending each messages. Finally, `mess` indicates a generic message from another process, and *Input* a process' set of incoming messages. Informally, the computation proceeds as follows. At the outset, a basic process sends some start messages to other basic processes. Before this, it informs the controller of the number of potential activities its messages have triggered in other processes, and changes its status to `active`. After sending off the initial messages, it sets its status to `passive`. Importantly, it informs the controller about such change of state. Then, the becomes passive, and sits waiting for incoming messages. On reception of some message (viz., $Input \neq \emptyset$), it handles them by performing some computation, and terminates the iteration by sending off some messages resulting from the computation to other basic processes. Again, it must inform the controller about the potential activities it is triggering with such messages, and of its newly acquired passive status, and increases its local time t ready for the successive iteration. This is expressed formally below in the pseudocode for a generic basic process `p`.

Protocol of processes.

```
p::
  C: Controller;
  status: {active,passive};
  t: integer;
  mess: message;
  Input: set of incoming messages;
begin
1.  send( M_p(0) + 1 ) to C;
2.  status = active;
3.  send M_p(0) messages to processes;
4.  t = 1;
5.  status = passive;
6.  send( -1 ) to C;
7.  while(true){
8.    if( Input ≠ ∅ ) then {
```

```
9.       mess = get_mess(Input);
10.      if( mess = STOP ) then break;
11.      status = active;
12.      handle(mess);
13.      send( M_p(t) ) to C;
14.      send M_p(t) messages to processes;
15.      t = t + 1;
16.      status = passive;
17.      send( -1 ) to C;
18.   } }
end.
```

The main role of the controller is to keep track of other process' activities. It does so sequentially, using variable *Act*. At the beginning, it waits the first message from some basic process as a signal that the system is launched. Then it repeated sums activities until the sum is found equal to zero and there are no more messages waiting to processes in the input queue. At that point, the system has terminated, and the controller informs all basic processes of that.

Protocol of agent-controller C.

```
C ::
  Act: integer;
  Input: set of integer;

begin
1.  Act = 0;
2.  while( Input = ∅ ) { }
3.  while(true){
4.    if( Input ≠ ∅ ) then Act = Act + get_mess(Input);
5.    if( Input = ∅ and Act = 0 ) then break;
6.  }
7.  send STOP to all;
end.
```

If we then let an instance of protocol `p` for the basic process p_i be denoted as `p_i`, the AB-algorithm for the distributed system **AB** can be presented as follows:

```
AB::
begin
   parallel {p_1} ...{p_n} {C}
end.
```

where the `parallel` operator means that all execution flows (threads) in the set of braces run in parallel.

3 Features and Correctness of the AB-algorithm

In this section we describe some of the characteristics of the base AB-algorithm. We use the property classification introduced in [8], which assumes the following categories for DTD-algorithms.

1. *Type of algorithm* – For example, a wave algorithm is a popular type of DTD algorithm.
2. *Necessary network topology* – For those algorithms where it is necessary to exploit the underlying topology of the network in order to detect termination correctly.
3. *Algorithm symmetry* – An algorithm is symmetric if each process runs an identical algorithm.
4. *Process knowledge* – For example, a process that requires information about the network in order to perform its duty, has special knowledge.
5. *Communication protocol* – Each algorithm assumes either synchronous or asynchronous communication.
6. *Message arrival* – Each algorithm assumes either first-in first-out (FIFO) message channels or no restrictions on the relative order of message delivery.
7. *Message optimality* – If an algorithm, in its worst case, uses the number of messages which researchers have proven to be a lower bound on the number of messages necessary to detect termination, it is considered message optimal.
8. *Fault tolerance* – If a system can detect termination when there are portions of the system that do not work as expected, it is considered fault tolerant.

Proposition 1.
The base AB-algorithm has the following features:

1. *the algorithm has the message-counting and credit/recovery type;*
2. *the basic network topology has no restrictions, apart that every process must be connected to the controller by duplex channels;*
3. *the algorithm is asymmetric;*
4. *no process needs special knowledge but the controller;*
5. *the communication protocol is synchronous;*
6. *there is no restriction on message arrival;*
7. *the algorithm is message optimal;*
8. *the algorithm is not fault tolerant.*

Sketch of the Proof.
(1) *Type of the algorithm.* Our algorithm has the message-counting type because the controller calculates the activity balance using the number of messages sent, as well as direct information about the activity of each process. The algorithm has the credit/recovery type too, because of an inversion of credit/recovery

course: the controller as a single debtor and all other processes as its creditors.

(2) *Necessary network topology.* The controller agent must definitely be connected with each processes by a duplex channel in order for it to be informed about their activity. Yet, there is no requirement for a special network topology of the basic processes.

(3) *Algorithm symmetry.* The AB-algorithm is asymmetric, because there is a special controller agent whose actions differ from others.

(4) *Process knowledge.* In our algorithm every process has just to know the controller, and vice versa; no other information is necessary for detecting termination.

(5) *Communication protocol.* Communication in the base algorithm must be synchronous. This means that messages are transmitted instantly and stored in channels until the are read.

(6) *Message arrival.* Message channels for processes in our algorithm need not be FIFO or of any other specific type, because the controller detects termination when its channel is empty.

(7) *Message optimality.* The AB-algorithm is message optimal because the number of input messages that every basic process handles is larger than the number of messages sent to the controller.

(8) *Fault tolerance.* Our algorithm is not fault tolerant: faulty processes may not send timely and correct information about their activity. ■

The correctness of the AB-algorithm is proved by the following proposition.

Proposition 2. *If the distributed system* **AB** *terminates, then the controller determines the termination moment correctly.*

Sketch of the proof.
The proof of the proposition follows from the fact that the value of variable *Act* becomes 0 with the empty input channel no earlier than the termination moment. Let $active(t)$ be the number of active processes and $Input(t)$ be the sum of all values in the *Input* channel at instant t. For every global time moment t it holds that $Input(t) + Act \geq active(t)$. This is because (1) each basic processes increases *Act* after its local termination, when it sends to the controller the number of the potential activities it triggered (lines 1,13); and (2) it decreases *Act* when it informs the controller about its passive status (lines 6,17). ■

We have also verified this system using the model checking tool SPIN [5]. We use the SPIN input language *Promela* for the above protocols, and expressed the correctness property for the controller in linear time temporal logic as follows.

$$G(Act = 0 \wedge Input = \emptyset \rightarrow \bigwedge_{p \in \mathbf{AB}} p.status = passive)$$

DTD-algorithms can be considered as knowledge-based programs [1]. In such programs process agents could act depending on their knowledge about world.

In particulary, our controller has to inform other processes about system termination only if it knows that the system stops. This knowledge property of the controller can be formulated as: if the controller detects the termination moment, then it *knows* exactly that every process is passive. This property is expressible in the logic of knowledge and time [1] as

$$G(Act = 0 \wedge Input = \emptyset \rightarrow K_C(\bigwedge_{p \in \mathbf{AB}} p.status = passive)).$$

This property could be verified by the knowledge model checking tools MCK[2] or VerICS [6]. Note, that the basic processes know that the system has terminated only after receiving the `STOP` message from the controller.

4 Modifications of the Base AB-Algorithm

In this section we suggest and analyze several modifications extensions of our base AB-algorithm, directly suggested by the classification from [8] we referred to and used in the previous section.

(1) Necessary network topology.
The basic network has no restriction on topology, but for the detection of termination every process has to be connected with the controller by a duplex channel. In order to drop the requirement of global connection with the controller, the basic network must provide a spanning tree of duplex channels for the exchange of messages between basic processes and the controller. Changes of network topology, which keep possibility of every process to communicate with the controller, do not affect correctness of the algorithm, but message optimality may be different, because some processes can be loaded with another's messages for the controller.

(2) Algorithm symmetry.
The AB-algorithm can be made symmetric by equipping every process with the logic to perform the controller actions. In this case, processes have to inform all network processes about their activity besides sending the basic messages. This has of course a big effect on network traffic, the number of messages grows, and the algorithm ceases to be message-optimal. This variant of the algorithm remains correct, that can be proved easily by induction on number of processes.

(3) Communication protocol.
We assume in the base algorithm that messages are transmitted instantly and stored in channels until they are read. Let us consider the asynchronous case, where messages are transmitted with a finite, yet indeterminate delay. If channels are of FIFO-type, then the base algorithm is still correct for the prior reasons, and requires no revision. If however the channels are not FIFO, then an additional control is required of each basic processes to detect termination.

In order to deal with this, let us add to every process a counter `MSent` which will be used to count the number of messages it sent. Now local time `t` is used for

counting a process' transitions to the passive status. The processes can receive `STOP` and `CONTROL` messages from the controller. The controller sends `CONTROL` message when it finds that there is no activity of other processes and no messages in its input channel. It suggests that system terminates but it is not sure because for the message delays some processes may not read messages for them. For this reason it has to control the number of messages in the system using information from other processes. In fact, the number `(MSent-t)` which each process p sends to the controller after receiving the `CONTROL` request, is nothing but the difference between the numbers of messages that p sent and handled at time `t`. If the controller finds that for each process p all the messages sent have been handled, i.e., the *control sum* of `(MSent-t)` from every p is zero, then it decides that the system has terminated. It is obvious that if this control sum is equal to zero then there are no messages transferred in the system. This fact and correctness of the base algorithm imply correctness of this asynchronous variant of AB-algorithm. With a little modification to our base algorithm, such a control sum is sufficient to detect termination. As before, the controller uses an activity counter *Act* to handle the calculations of the control sum efficiently. This is not strictly necessary, but it simplifies the protocol significantly. The modified version of our base AB-algorithm is described below.

Protocol of processes.

```
p::
  C: Controller;
  status: {active,passive};
  t, MSent: integer;
  mess: message;
  Input: set of incoming messages;

begin
1.  send( M_p(0) + 1 ) to C;
2.  status = active;
3.  send M_p(0) messages to processes;
4.  t = 1;
5.  status = passive;
6.  send( -1 ) to C;
7.  while(true){
8.   if( Input ≠ ∅ ) then {
9.      mess = get_mess(Input);
10.     if( mess = STOP ) then break;
11.     if( mess = CONTROL ) then send( MSent - t ) to C;
12.     if( mess from OtherProcess ) {
13.         status = active;
14.         handle(mess);
15.         send( M_p(t) ) to C;
16.         send M_p(t) messages to processes;
17.         MSent = MSent + M_p(t);
```

```
18.          t = t + 1;
19.          status = passive;
20.          send( -1 ) to C; }
21.   } }
end.
```

For simplicity we allocate a special channel *Contmessages* for the control sums from the basic processes. The controller must know the number of network processes in order to wait for all messages on this channel. The protocol of the controller is below, where `n` is the number of processes in the network.

Protocol of agent-controller C.

```
C ::
  Act, ContSum, i: integer;
  Input, Contmessages: set of integer;

begin
1.  Act = 0;
2.  while( Input = ∅ ) { }
3.  while(true){
4.    if( Input ≠ ∅ ) then Act = Act + get_mess(Input);
5.    if( Input = ∅ and Act = 0 ) then {
7.      ContSum = 0;
8.      clear(Contmessages);
9.      send CONTROL to all;
10.     while( Input = ∅ and |Contmessages| < n ) { }
11.     if( Input = ∅ ) then
12.        for( i=0; i < n; i=i+1 )
13.          ContSum = ContSum + get_mess(Contmessages);
14.     if( Input = ∅ and ContSum = 0 ) then break; } }
15. send STOP to all;
end.
```

(4) Dynamics.
The basic system can be dynamic, i.e., basic processes can appear and disappear whilst the termination detection protocol is running. In this case, the base AB-algorithm remains correct under the assumption that a basic processes can disappear only when it is in passive state, its set of input messages is empty, and sending messages to disappeared processes is forbidden. The reason for the correctness is that if a disappearing process satisfies these conditions then its extinction does not affect activity of other process and counting this activity.

(5) Hierarchical networks.
In a hierarchical network every basic subprocess sending messages communicates to the controller by itself. A basic process is passive if and only if all its subprocesses are passive. It is obvious that the base algorithm remains correct due to autonomy of subprocess communication with the controller.

We remark that the AB-algorithm remains correct with respect to any combination of the above extensions, except for the combination of non-FIFO asynchronous communication and dynamics. This combination fails because in non-FIFO asynchronous mode the controller has to know the number of system processes.

5 Conclusion

We have presented a new Activity Balance algorithm for the distributed termination detection problem. We have proved the algorithm's correctness and illustrated some of its properties. We analyzed and investigate several significant extensions to the algorithm, with the exception of the extensions to a fault-tolerant algorithm, which we leave for future work. In the future, we also plan to verify knowledge properties of the AB-algorithm and its extensions using model checking tools for logics of knowledge in multi-agent systems. This research is part of the study of DTD-algorithms in the context of reasoning about knowledge.

Acknowledgements: I would like to thank my colleague Igor Anureev, as well as Vladimiro Sassone for help and discussions.

References

1. **Fagin R., Halpern J.Y., Moses Y., Vardi M.Y.** Reasoning about Knowledge. — London: MIT Press, 1995. — 477 p.
2. **Gammie P., van der Meyden R.** *MCK: Model Checking the Logic of Knowledge* // Proc. of 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004. LNCS Vol. 3114, 2004, pp 479-483.
3. **Garanina N., Sidorova E., Bodin E.** *A Multi-agent Approach to Unstructured Data Analysis Based on Domain-specific Onthology* // Proc. of the 22nd International Workshop on Concurrency, Specification and Programming, Warsaw, Poland, September 25-27, 2013. CEUR Workshop Proceedings, Vol-1032, P. 122-132
4. **Huang, S.** *Detecting termination of distributed computations by external agents.* // In: IEEE Nineth International Conference on Distributed Computer Systems, pp. 79-84.
5. **Holzmann G. J.** The Spin Model Checker: Primer and Reference Manual.// Addison Wesley Pub, 2003. P. 608
6. **Kacprzak M., Nabialek W., Niewiadomski A., Penczek W., Plrola A., Szreter M., Wozna B., Zbrzezny A.** *VerICS 2007 - a Model Checker for Knowledge and Real-Time* // Fundam. Inform. 85(1-4): 313-328 (2008)
7. **Kumar, D.** *A class of termination detection algorithms for distributed computations* // Proc. of the Fifth Conference on Foundations of Software Technology and Theoretical Computer Science. LNCS, Vol. 206, 1985, pp 73-100.
8. **Matocha J., Camp T.** *A taxonomy of distributed termination detection algorithms* // The Journal of Systems and Software, 1998, Vol. 43, P. 207-221
9. **Mattern, F.** *Algorithms for distributed temination detection.*// Distributed Computing 2 (4), 161-175.

10. **Mattern, F.** *Experience with a new distributed termination detection algorithm.* // In: Proceedings of the Second International Workshop on Distributed Algorithms, pp. 127-143.
11. **Mattern, F.** *Global quiescence detection based on credit distribution and recovery* // Inform. Process. Lett. 30 (4), 1989, P. 195-200.
12. **Rokusawa, K., Iciyoshi, N., Chikayama, T., Nakashima, H.** *An effcient termination detection and abortion algorithm for distributed processing systems* // In: Proceedings of the International Conference on Parallel Processing, pp. 18-22.