Distributed 2-Vertex Connectivity Test of Graphs Using Local Knowledge

Brahim Hamid, Bertrand Le Saëc, and Mohamed Mosbah

Abstract—The vertex connectivity of a graph is the smallest number of vertices whose deletion separates the graph or makes it trivial. This work is devoted to the problem of vertex connectivity test of graphs in a distributed environment based on a general and constructive approach. The contribution of this paper is threefold. First, using a pre-constructed spanning tree of the considered graph, we present a protocol to test whether a given graph is connected, undirected graph denoted by $G$, where $M$ is the number of its edges, $N$ the number of its nodes and $\Delta$ is its degree, our algorithms need the following requirements: The first one uses $O(\Delta \times N^2)$ steps and $O(\Delta \times \log \Delta)$ bits per node. The second one uses $O(\Delta \times N^2)$ messages, $O(N^2)$ time and $O(\Delta \times \log \Delta)$ bits per node. Furthermore, the studied network is semi-anonymous. Only the root of the pre-constructed spanning tree needs to be identified.

Keywords—Distributed computing, fault-tolerance, graph relabeling systems, local computations, local knowledge, message passing system, networks, vertex connectivity.

I. INTRODUCTION

DISTRIBUTED connectivity test of a graph has gained much attention recently. Indeed, many distributed applications such as routing or message diffusion assume that the underlying network is connected [1], [11], [10], [12]. Connectivity information is also useful as a measure of network reliability. This information can be used to design distributed applications on unreliable networks. For example, testing and then preserving $k$-connectivity of a wireless network is a desirable property to tolerate failures and avoid network partition [1]. In this paper, we propose a general and constructive approach based on local knowledge to test whether a given graph modeling a network is 2-connected in a distributed environment: A network is represented as a connected, undirected graph denoted by $G = (V, E)$ where a node in $V$ represents a process and an edge in $E$ represents bidirectional communication link. Therefore, our work may be used as a core of protocols using the connectivity as input information.

The Problem: Consider a system modeled by a graph $G = (V, E)$. We denote by $N$, where $N \geq 2$, the number of the nodes of $G$ and by $M$ the number of its edges. The degree of $G$ is denoted by $\Delta$. The vertex connectivity of a graph $G$, or simply its connectivity, is the minimum size of a node set of $G$ whose deletion separates, or disconnects, the graph $G$ or makes it trivial. In many distributed applications, the connectivity of the underlying network is considered as an input data. In dynamic networks, where nodes may disappear resulting from failures, such information becomes not consistent. Therefore, the design of a protocol to test whether a graph satisfies some connectivity improves the consistency of the designed applications. Given a graph $G$, the test of the 2-vertex connectivity of such a graph, or simply 2-connectivity, deals with the problem of testing whether $G$ is 2-connected or not. A protocol that solves such a problem responds to the question: The number of nodes that must be deleted in order to disconnect a graph is at least 2. Furthermore, we study this problem in a distributed setting using only local knowledge. That is, each node “communicates”, or exchanges information, only between neighboring nodes and each computation acts on the states of neighbors.

Previous Works: In the late 1920s, Menger [12] studied the connectivity problem and some related properties. Since then many results have been obtained in this area. The best known works about the $k$-vertex connectivity test problem may be summarized in [12], [9], [3], [2], [5], [4]. In [9] an algorithm to test the 2-connectivity of graphs is given. This is a depth first search based algorithm with a time complexity of $O(M + N)$. Thereafter, the computation of the vertex connectivity is reduced to solve a number of max-flow problems. For the first time this approach was applied in [3]. The computation of the maximum flow is used as a basic procedure to test the vertex connectivity. The remaining works try to reduce the number of calls to max-flow procedure using some knowledge about the structure of the graph. The time complexity of the presented algorithms is bounded by $O(k \times N \times M)$. All of these works are presented in a centralized setting assuming global knowledge about the graph to be examined.

Our Model: To encode distributed algorithms we will use two general models: The first one is the graph relabeling systems (GRS) [6] called here the local computations model. In this model, the local state of a process is encoded by labels associated with the corresponding node. At each step of the computation on a node, labels are modified in a ball of neighbors around this node. The modification is given by rewriting rules of the relabeling system. These rules can be applied in any order or even concurrently on disjoint subgraphs. In this work, we consider only relabeling between two neighbors. We assume that each node distinguishes its neighbors and knows their labels. The execution time of a computation is the length of the corresponding relabeling sequence. A measure of a space complexity of a distributed algorithm is the size of the required labels.

The second one is the asynchronous message passing model $(MPS)$[10]. In this model, each process (node of the graph)
has its own local memory, with no shared global memory. Processes communicate by sending and receiving messages through existing communication links (edges of the graph). Networks are asynchronous in the sense that processes operate at arbitrary rates and message transfer delays are unbounded, unpredictable but finite. However, we assume that message orders are preserved.

Our Contribution: This paper deals with the 2-vertex connectivity problem using a set of procedures to show formally and easily the behavior of the presented algorithm. Let $G$ be a graph with a distinguished node $v_0$ and let $T$ be a spanning tree of $G$ rooted at $v_0$. It is known that for a graph with a distinguished node, a distributed computation of a spanning tree can be easily performed [7]. Therefore, the assumption of a pre-constructed spanning tree is not a loss of generality. The main idea of the 2-connectivity test algorithm is based on a traversal of the tree starting from the root. Each visited node $v$ tries to test whether the subgraph induced by its possible deletion remains connected. That is, a spanning tree of the graph deprived of $v$ can be built. Therefore, the original graph is 2-connected if such a construction can be made for all the nodes of $G$. In the opposite case, the graph is not 2-connected. This approach is based on an occurrence of the the Menger theorem [12]. Our main contribution is to give a protocol to test the 2-connectivity of graphs in a distributed way using only local knowledge. Moreover, this algorithm does not need a completely identified network. Only the root of the pre-constructed spanning tree needs to be identified: the network is semi-anonymous.

Then, we give two possible implementation of our protocol in the two previous models. For a given graph $G$, our GRS algorithm uses $O(\Delta \times N^2)$ time and $O(\Delta \times \log \Delta)$ bits per node. The other one (MPS) uses $O(\Delta \times N^2)$ messages, $O(N^2)$ time and the same space requirements as the GRS algorithm. Additionally, our protocol may be used as a brick to deal with the $k$-vertex connectivity test of graphs.

The rest of the paper is organized as follows. The model used to encode distributed algorithms is explained in Section 2. In Section 3, we introduce the notion of procedure using an example of the distributed spanning tree computation. In Section 4, we present our protocol to deal with the problem of the 2-vertex connectivity test of graphs. In the following section, we present an encoding of this protocol in the local computations model. We discuss in Section 6 the implementation of this protocol in the asynchronous message passing model. Finally, Section 7 concludes the paper with short discussion about future works.

II. PRELIMINARIES

A. Graphs

A graph is a collection of nodes $V$ where some of them are connected by edges $E$. So a graph is a couple $(V,E)$ where $E \subseteq V^2$. We use sometimes the notations $V_G, E_G$ to denote respectively $V, E$ where $G = (V,E)$. Node $u$ is a neighbor of node $v$ if $(u,v) \in E$. A path $p$ in $G$ is a sequence $(v_0, ..., v_l)$ of nodes such that for each $i < l, (v_i, v_{i+1}) \in E$. The integer $l$ is called the length of $p$. A path $P = (v_0, ..., v_l)$ is simple

III. SPANNING TREE PROCEDURE

A tree $T = (V_T, E_T)$ is a connected graph that have no cycle and an elected node $v_0$ called the root. We denote by $\text{Sons}(v)$ the set of the sons of $v$. In the sequel, we assume that the list “Sons” of $v$ is ordered. The first son of $v$ will be called “Preferred” son of $v$. A node $u \in V_T$ with no son is called a leaf of $T$. Then, a spanning tree $T = (V_T, E_T)$ of a graph $G = (V_G, E_G)$ is a tree such that $V_T = V_G$ and

1 $\#M$ denotes the cardinality of the set $M$. 
\( Et \subseteq Ec \): The tree can be defined also by \( T(\text{Father}(v), v) \). We denote by \( T_u \) a tree with root \( v \). We denote by \( T(u)\setminus v \) the maximal subtree of \( T \) that contains the node \( u \), but not \( v \).

We present here a distributed algorithm to compute a spanning tree \( T \) of a graph \( G \), encoded in the local computations model. Since it will be used by our algorithm, it is presented as a \textit{procedure} with parameters. This notion of procedure is similar to the \textit{“interacting components”} used in [8]. The \textit{spanning tree procedure} is referenced as follows:

\[
\text{STP}(G,T; v_0, \text{Stage}; x,y; \text{Father}; \text{Sons}), \quad \text{where } G \text{ is the treated graph and the procedure builds a spanning tree } T \text{ rooted at } v_0. \text{ The structure of } T \text{ is stored locally in each node using the labels } \text{Father} \text{ and } \text{Sons}. \text{ Initially, the value of the label } \text{Stage} \text{ at each node } v \in G \text{ in } X. \text{ At the end of the computation, the value of } \text{Stage} \text{ at all the nodes is } Y. \text{ The following graph relabeling system describes an encoding of the spanning tree procedure.}
\]

### Spanning Tree Procedure: \text{STP}(G,T; v_0, \text{Stage}; x,y; \text{Father}; \text{Sons})

- **Input:** A graph \( G = (V,E) \) and \( v_0 \) the chosen root.
- **Labels:**
  - \( \text{Stage}(v) \in \{X,Y,W,A\} \)
  - \( \text{Father}(v) \in \emptyset \)
  - \( \text{Sons}(v) \in \emptyset \)
  - \( \text{Terminated}(v) \in \emptyset \)
- **Initialization:** \( \forall v \in V, \text{Stage}(v) = X. \)
- **Results:** A spanning tree \( T = (V,ET) \) of \( G \) with root \( v_0 \) such that \( \forall v \in V, \text{Stage}(v) = Y. \)
- **Rules:**
  - **STP1:** The node \( v_0 \) starts the computation
    - **Procedure:**
      - Stage\((v_0) = X \)
  - **STP2:** Spanning rule acting on 2 nodes \( v, w \) where \( w \) is not yet in the on-building tree
    - **Procedure:**
      - \( v \in B(u) \) and Stage\((v) = WA \)
      - Father\((v) = w \)
      - Sons\((v) = \emptyset \)
      - Terminated\((v) = \emptyset \)
      - \( \forall v \in V, \text{Stage}(v) = X. \)
  - **STP3:** Node \( v \) discovers its neighbors already included in the tree
    - **Procedure:**
      - Stage\((v) = WA \)
      - If \( B(u), \text{Stage}(u) = WA \) and \( w \notin \text{Included}(u) \)
      - Included\((v) = \text{Included}(v) \cup \{ w \} \)
      - Terminated\((v) = \text{Terminated}(v) \cup \{ w \} \)
  - **STP4:** Node \( v \) finishes locally the computation of a spanning tree
    - **Procedure:**
      - Stage\((v) = WA \)
      - If \( \text{Father}(v) \notin \emptyset \) then
        - Father\((v) = Y \)
        - Terminated\((\text{Father}(v)) = \text{Terminated}(\text{Father}(v)) \cup \{ v \} \)

---

Here, Father\((v) = \emptyset \) means that \( v \) has no defined father. The computation finishes when Stage\((v_0) = Y. \) In this case, all the nodes \( v \) also satisfy Stage\((v) = Y. \) Obviously we have a spanning tree of \( G \) rooted at \( v_0 \) defined by the third components and the fourth components of the labels of the nodes. The root of the spanning tree is then the unique node with its father equal to \( \perp. \) Therefore, we claim the following:

**Property 1:** For a graph \( G = (V,E) \), when a distinguished node \( v_0 \) is labeled Stage\((v_0) = Y. \), \( \#E + \#V + 1 \) rules have been applied. Then, the spanning tree \( T \) of \( G \) rooted at \( v_0 \) has been built.

### IV. 2-Vertex Connectivity Test of Graphs

In this section we present a protocol to test whether a given graph \( G \) is 2-connected. Since a 2-connected graph is a graph without any cut-node, we explore all the nodes of \( G \). Each visited node is tested to know if it is or not a cut-node. That is an occurrence of the Menger theorem [12]:

**Proposition 2:** Let \( T \) be a spanning tree of a graph \( G \).

Then the graph \( G \) is 2-connected if \( \forall v \in V_G, \exists E \subseteq Ec \setminus ET \text{ such that } \langle V_{G,v} , ET_{G,v} , E \rangle \text{ is a spanning tree of } \langle V_{G,v} , Ec_{G,v} \rangle \text{ with } \#E \leq \#SON(v). \)

Our distributed 2-vertex connectivity test algorithm consists of the two following rounds:

1. The computation of the spanning tree called \textit{Investigation tree}, denoted by \textit{Inv}_{T}, of \( G \) with root \( v_0 \).
2. Exploration of \textit{Inv}_{T} to identify cut-nodes of \( G \). In round one, we use an algorithm as described in the previous section. This procedure constructs a spanning tree \textit{Inv}_{T} of a given graph \( G \) rooted at \( v_0 \) with local detection of the global termination. It means that \( v_0 \) detects the end of the spanning tree computation of \( G \). In round two, we explore the tree \textit{Inv}_{T} using \textit{“depth-first trip”} \[10\]. When the trip reaches a node \( v \), \( v \) does the following:

   1. \textit{configure}. Node \( v \) constructs a spanning tree \( T \) of \( G \) rooted at itself and initializes the set of labels erasing the trace of the last test. The aim of this procedure is to prepare its test.
   2. \textit{disconnection}. \( v \) disconnects itself, it will be labeled \textit{Stage}(v) = D.
   3. \textit{reconnection}. Node \( v \) orders to its preferred son to construct a spanning tree of the graph \( G' = (V_{G,v}, Ec_{G,v}) \) rooted at the preferred son of \( v \). Node \( v \) is not a cut-node if its preferred son succeeds this task.

The algorithm terminates according to the following cases:

(a) if some node fails the reconnection phase, (b) after the exploration of all nodes of \( G \) without the previous case. In the last case, \( v_0 \) states that \( G \) is 2-connected. As we shall see, the aim of the configuration phase represented above as \textit{configures} is achieved by the current tested node which builds an auxiliary tree \( T \) of \( G \) adding the simulation of the disconnection
of the current explored node \( v_d \). In the next section, we present an implementation of the reconnection procedure. It is the main procedure of our protocol.

Given a node \( v_d \) of a spanning tree \( T \) of a graph \( G \), the reconnection procedure builds, if possible, a spanning tree of \( G \setminus v_d \) rooted at \( r \). It works as follows: Let \( T \) be a spanning tree of \( G \). We denote by \( T' \) the “on-building” tree of \( G \setminus v_d \) rooted at \( r \). This procedure is composed of the following steps:

1) computes the base of the tree. This base of \( T' \) is the subtree of \( T \setminus v_d \) rooted at \( r \). It is identified using a marking procedure (see below).

2) **search extension.** Using a “depth-first trip” \([10] \) on \( T' \) each node \( v \) is visited until one of the neighbor of \( v \) is not yet included in the current tree \( T' \).

3) **connection found.** The visited node \( v \) finds a neighbor \( u \) that is not in the “on-building” tree. Such a node is called a connection entry and \( v \) is named a connection node. Therefore, the edge \((u,v)\) is added to \( T' \). The subtree of \( T \) containing \( u \) is reorganized in such a way that \( u \) becomes its root using the root changing procedure (see below). Then, this subtree is included in \( T' \) and its nodes are marked as nodes of \( T' \). Now, the search of isolated subtrees is restarted on the extended \( T' \).

Finally, \( r \) detects that it had extended as much as possible its tree and the procedure is terminated.

V. ENCODING IN THE LOCAL COMPUTATIONS MODULE

The protocol is implemented in both graph relabeling systems and asynchronous message passing model using a set of procedures. Here we present the first implementation. In Section 6, we discuss one of the procedures used in the second implementation.

A. Reconnection Procedure: \( RP(G, T, T', r, v_d) \)

We start with the main procedure: the reconnection procedure. It is composed of nine rules and uses two procedures: the marking procedure and the root changing procedure. In the sequel we present a short description of these two procedures. The reconnection procedure is more detailed.

**Marking Procedure** \( MP(T; v_0, Stage, X, Y) \): This procedure allows us to change the label \( Stage \) of all the nodes of \( T \) from \( X \) to \( Y \) Therefore, if the \( MP \) is applied to a tree \( T = (V, E) \) rooted at \( v_0 \), \( v_0 \) will detect that all the nodes of \( T \) are marked \( Y \) after the application of \( 2\#V \) rules.

**Root Changing Procedure** \( RCP(T; u) \): This procedure makes \( r \) the new root of the tree \( T \). Hence, after the application, in the worst case, of \( 2\#V \) rules node \( u \) becomes the new root of \( T \).

The two following Lemmas show the behavior of the reconnection procedure.
Lemma 3: Let $v_d$ be a cut-node of a graph $G=(V,E)$. Then the reconnection procedure fails to construct a spanning tree $T'$ of $G \setminus v_d$.

Sketch of the Proof. By contradiction. We suppose that the reconnection procedure will succeed to construct a spanning tree of $G \setminus v_d$. This means that $G \setminus v_d$ remains connected. We show that after the execution of such a procedure, some son of $v_d$ is labeled $A$ which contradicts our assumption.

Lemma 4: Let $v_d$ be not a cut-node of a graph $G=(V,E)$. Then the reconnection procedure will succeed to construct a spanning tree $T'$ of $G \setminus v_d$ rooted at the preferred son $r$ of $v_d$.

Sketch of the Proof. The proof is by induction on the size of $\#Sons(v_d)$. After the disconnection of such a node, we have $\#Sons(v_d)$ sub-trees. We must show that after such event, the reconnection procedure will re-connect such sub-trees to construct a spanning tree $T'$ of $G \setminus v_d$.

B. 2-Connectivity Test Algorithm

In this part we present our protocol referred to in the rest of the paper as 2VCT algorithm. This algorithm uses three procedures: The spanning tree procedure to construct the investigation tree (round 1), the configuration procedure to encode phase (1) of round 2. Phase (3) of this round is achieved using the reconnection procedure presented above. Now, we present a short description of the configuration procedure. Then, we give an example of 2VCT algorithm’s run.

Configuration Procedure ($CP(G,T;v_d)$): Given a graph $G=(V,E)$, this procedure allows to build a spanning tree $T$ of $G$ rooted at $v_d$. In addition, all the required initializations of the labels of $G$, used in the reconnection procedure, are achieved here. This procedures uses an extended version of the marking procedure and an extended version of the spanning tree procedure. Moreover, $v_d$ prepared its disconnection. So each of its sons sets its label “Father” to $\bot$. Thus, the cost of the configuration procedure is $O(#E + #V)$ time when applied to a graph $G$.

2VCT Algorithm: To store locally the investigation tree $Inv_T$, each node has the following labels in addition to the labels used during the execution of the other procedures: $InvStage(v)$, $InvFather(v)$, $InvSons(v)$.

Let $G=(V,E)$ be a graph and $v_0 \in V$ be a node to launch and supervise the test process. At the start of the computation, we have $\forall v \in V, InvStage(v) = N$. The node $v_0$ starts the computation: It builds a spanning tree $Inv_T$ of $G$ rooted at $v_0$. Subsequently, all the vertices of $G$ are such that $InvStage = A$. Since the test scheme is based of the use of a “depth-first trip” exploration on the tree $Inv_T$, the root $v_0$ is the only one able to start the trip. Each node ready to be examined switches its label to $On_{Test}$.

The examination phase of the node $v_d$ consists of the test of whether $v_d$ is or is not a cut-node of the graph $G$. It proceeds as follows. The node $v_d$ labeled On$_{Test}$ constructs a spanning tree of the graph $G$, using the configuration procedure. At the end of this procedure, $v_d$ is labeled Stage($v_d$) = D and its preferred son Preferred($v_d$) tries to build a spanning tree of the graph $G \setminus v_d$ using the reconnection procedure. As we shall see in the next paragraph, eventually $v_d$ decides if it’s or not a cut-node of $G$.

If $v_d$ is not a cut-node, it transfers the label On$_{Test}$ to one of its not yet tested sons (“InvSons”). Since the leaf nodes of the tree $Inv_T$ are not “cut-nodes”, the “chosen” son must not be a leaf. Note that the leaf node must be avoid from the list “InvSons” of its father in the tree $Inv_T$.

Eventually a node $v_d$ finishes the test of all the nodes in its subtree (InvSons($v_d$) = $\emptyset$). So it informs its father that the computation is over. Now $v_d$ is avoided from the list InvSons of its father which then can continue the exploration choosing one of its not yet tested son. Furthermore, only the root $v_0$ detects the end of the “trip” when all its sons are over. It means that all the nodes are tested and succeeded. Then, $v_0$ affirms that the graph $G$ is 2-connected.

In the opposite case ($v_d$ is a cut-node), the reconnection procedure fails. So the tested node $v_d$ will be labeled Stage($v_d$) = KO. Then $v_d$ sets its label InvStage($v_d$) to Over. There are two possibilities: First, if $v_d \neq v_0$ then it informs its father in $Inv_T$ and so on until the root $v_0$. Second, if the reconnection procedure fails about the root $v_0$. So, $v_0$ is a cut-node. It stops the test procedure (InvStage($v_d$) = KO). In the two cases, $v_0$ affirms that the graph $G$ is not 2-connected.

C. Overall Proof of Correctness and Complexity Analysis

Now, we show the correctness of the 2VCT algorithm using a scheme based on the procedures properties. Assume that when the procedures are executed in the relabeling part of some rule: They are executed as atomic steps with the two following consequences:

- Each procedure achieves its expected task,
- Each procedure terminates during the execution of the rule referencing it.

The analysis is closed with time and space complexity measures. First, we claim the following:

Fact 5: If the graph $G=(V,E)$ contains at least one cut-node then it is not 2-connected. Otherwise, if it doesn’t contain any cut node then it is 2-connected.

Theorem 6: The 2VCT algorithm presented above implements a distributed test of the 2-vertex connectivity of a graph.

Sketch of the Proof. The correctness proof is by induction on the number of nodes of $G$ and the results of Fact 5, Lemma 3 and Lemma 4. This is based also on the used exploration technique.

For the time complexity, the worst case of the execution of the 2VCT algorithm corresponds to the case when the reconnection procedure is applied to each node. For the space complexity, each node $v$ is labeled $L(v)$ using the two following components:

- $B(v)$, Included($v$), Terminated($v$), Sons($v$), Feedbacks($v$), To Explore($v$), Potential($v$).
To Ignore, Treated(v), InvSons(v),
(2) Stage(v), Father(v), InvStage(v), InvFather(v).

Thus, to encode the first component of a label, every node needs to maintain subsets of its neighbors as descendants. By taking into account the other variables used in the second component of labels, we can claim that each node \( v \) needs \( O(\deg(v) \times \log \deg(v)) \) bits.

The following result completes the analysis.

**Theorem 7:** Given a graph \( G = (V, E) \). The 2VCT algorithm computes the test of the 2-vertex connectivity of \( G \) in \( O(\deg(G) \times \# V^2) \) time using \( O(\deg(G) \times \log \deg(G)) \) bits per node.

VI. ASYNCHRONOUS MESSAGE PASSING MODEL

Since computations used in our protocol are based on local information, the implementation of the presented modules and then the 2VCT algorithm is implemented in the asynchronous message passing model using a transformation (translation) from the previous one adding simple changes. The proofs of correctness and the complexities analysis are also based on those of the local computations model. Here, we give an example of an implementation of the spanning tree procedure, as presented in Section 3, to illustrate such a transformation.

We used the same variables as those of GRS model. The initiator (the root) \( v_0 \) starts the computations broadcasting a \(<st,fok>\) message to all its neighbors. At any step of the computation, when a node \( v \), not yet in the tree (\( Stage(v) \neq WA \)) receives a \(<st,fok>\) message from its neighbor \( w \), node \( v \) includes itself in the tree by changing its \( Stage \) to \( WA \). Moreover, in the same time, \( Father(v) \) is set to \( w \), set “Included” is now composed of \( w \) and set “Terminated” is initialized to \( \emptyset \). Finally, \( v \) informs \( w \) to add it in its set of sons sending the \(<st,son>\) message. At the reception of such a message, \( w \) adds \( v \) in both its sets of sons and in the set of nodes that are included in the tree (“Included”). When \( v \) finds all its neighbors already included in the tree, it detects that it has locally terminated its computation, then it informs its father. The computation finishes when all the nodes \( V \) are such that \( Stage(v) = V \) which means that they terminated locally their computations. And obviously we have a spanning tree of \( G \) rooted at \( v_0 \) defined using the same components as the one computed in the local computations model.

VII. CONCLUSION AND FUTURE WORKS

This work deals with the test of the 2-vertex connectivity of graphs in a distributed setting using local knowledge. We present a new formalization of such a problem using a constructive and a general approach. It is based on an occurrence of Menger’s theorem and its relation with the possible construction of a spanning tree. The protocol is encoded in the local computations model and implemented in the message passing model using a set of procedures. Given a graph \( G = (V, E) \) with degree \( \Delta \) and \( N \) nodes: The first algorithm requires \( O(\Delta \times N^2) \) steps and \( O(\Delta \times \log \Delta) \) bits per node. For the second one, without combining some phases to reach better some constants, it achieves correctly the test of the 2-vertex connectivity of \( G \) in \( O(N^2) \) time using \( O(\Delta \times N^2) \) messages and the same space requirements as the first one.

Thus, our work has an extra benefit: Algorithms encoded in the local computations model may be implemented in the asynchronous message passing model. Furthermore, the transformation guarantees the following: The proofs and the complexities measures used in the first one allows to deduce those of the second one. Therefore, a small bridge is proposed between these two models.

For the \( k \)-vertex connectivity test, we conjecture that we can use the previous procedures to generalize our algorithm to test the \( k \)-vertex connectivity of graphs in polynomial time. Intuitively, we can reduce this problem to the case of \( k = 2 \) followed by an incremental test procedure. One interesting application of the distributed \( k \)-vertex connectivity testing is the measure of a network fault-tolerance. In fact, the quality of the reachability of any couple of nodes in an unreliable network, and hence their communication, will depend on the number of paths between these nodes. If such paths are already computed, it is essential to maintain them while crash or disconnection of certain nodes occur.

REFERENCES