Distributed Problem Solving In Spite Of Processor Failures

K.V.S. Ramarao

Technology Resources
Southwestern Bell Company
550 Maryville Centre Drive
St. Louis, MO 63141

S. Venkatesan*

Computer Science Program
University of Texas at Dallas
Richardson, TX 75083-0688
venky@utdallas.edu

Abstract

Processor failures not leading to a network partition are considered and the issue of computing associative functions *in spite of processor failures is addressed. An intuitive and fundamental result formally proved here is that* failure detection *and* computing associative functions *are equivalent in faulty networks – one can be performed if and only if the other can be performed. Protocols and impossibility results in various system models are presented in the context of computing associative functions and solving topological problems.*

1 Introduction

It is important to deal with component failures in a distributed system, and protocols operating in a distributed environment must be able to cope with failures. There are several types of failures studied in the literature. Among these, *fail-stop* failure is the most benign failure – a component fails by simply stopping. One approach to the design of *fault-tolerant* protocols is to design the protocols resilient to fail-stop failures and then implement the fail-stop model in other failure environments. For example, see [10] for implementing fail-stop failures in a Byzantine failure environment. We consider only fail-stop failures from now on.

The problem of coping with processor failures has been extensively studied in the context of basic problems such as reaching consensus, implementing global clocks, etc., and several interesting results and impossibility proofs have been obtained. Fischer et al. [6] consider the consensus problem in an asynchronous system with a single faulty processor. The problem of reaching agreement among the processors is a fundamental problem of theoretical and practical importance in distributed systems. In an asynchronous system, the consensus problem cannot be solved in the presense of one faulty processor [6]. Also note that in solving any non-trivial problem, the consensus problem is implicitly solved. Thus, it is impossible to cope with processor failures in asynchronous systems.

The contributions made in this paper are two-fold. The first part of the paper addresses the issue of the existence of protocols to compute associative functions in the presence of processor failures not leading to a network partition, and protocols and impossibility results are presented. The second part explores the possibility of solving topological problems. In presenting the results, we assume that processors know the indentities of their neighbors, processor failures may occur at any time, a failed processor never recovers during the execution of the protocol, the links connecting two non-faulty processors are non-faulty, and the distributed system remains connected in spite of the processor failure(s).

The organization of the paper is as follows: The model of a distributed system is presented in Section 2. In Section 3, some preliminary results are presented. In Section 4, the notion of fault-tolerant computation of associative functions, the notion of failure detection, and a proof of their equivalence are presented. Protocols for failure detection, when possible, and impossibility results in the other cases are presented in Section 5. In Section 6, some discussion on solving topological problems in spite of processor failures is given. Section 7 concludes the paper.

2 System Definition

A distributed system is modeled by an undirected graph G = (V,E), where V represents the set of *pro-*

*This research was supported in part by NSF under Grant No. CCR-9110177 and by the Texas Advanced Technology Program under Grant No. 9741-036.

cessors and E represents the set of bidirectional *communication links*. Processors at the two ends of a link are said to be neighbors of each other. A path is a sequence of links. The distributed system is said to be connected if a path exists between every pair of processors. Communication between the processors is by message-passing only.

A protocol or a distributed algorithm is a collection of local algorithms at each participating processor. Each local algorithm consists of several steps (or events). In a step, a processor reads a message received from a neighbor, performs some local computation, changes its processor state, and sends a message (or several messages) to some (or all) of its neighbors. A processor may take a step even if it does not receive a message. Each processor has a unique *id* (of length $O(\log|V|)$ bits) associated with it, and each processor knows its own *id*, the *ids* of its neighbors and the links that lead to these neighbors.

The characteristics of a distributed system are captured by the following five types of system parameters in a distributed processing system [5].

1. Processors
   - *Asynchronous.* A processor can wait an arbitrarily long but finite amount of real time between two of its own steps.
   - *Φ-Synchronous.* There is a constant $\Phi \geq 1$ such that in any time interval in which some processor takes $\Phi + 1$ steps, every non-faulty processor must take at least one step.
2. Communication
   - *Asynchronous.* Message delivery time is arbitrarily long but finite.
   - *Δ-Synchronous.* There is a constant $\Delta \geq 1$ such that every message is delivered within Δ real-time steps.
3. Message Order
   - *Asynchronous.* Messages can be delivered out of order.
   - *Synchronous.* If p sends a message m_1 to r at real time t_1 and q sends a message m_2 to r at real time t_2 and if $t_2 > t_1$, then r receives m_1 before m_2 (p, q, and r are not necessarily distinct).
4. Transmission Mechanism
   - *Point-to-point.* In an atomic step, a processor can send a message to at most one processor.
   - *Broadcast.* In an atomic step, a processor can broadcast a message to all of the processors.
5. Receive/Send
   - *Separate.* In an atomic step, a processor cannot both receive a message and then send a (possibly different) message (or messages).
   - *Atomic.* Receiving a message and sending a (possibly different) message are part of the same atomic step.

The above mentioned five system parameters yield 32 different types of systems. Among these, the following four cases are the only (minimal) cases in which consensus is possible in spite of (fail-stop) failures [5]:

1. Synchronous communication and atomic receive/send (and broadcast transmission for multiple processor failures).
2. Synchronous processors and synchronous communication.
3. Synchronous processors and synchronous message order.
4. Broadcast transmission and synchronous message order.

For the first three cases, we show that it is possible to cope with processor failures in the context of computing associative functions while in the fourth case, we show that it is impossible to cope with the failure of one processor in computing associative functions. Thus, computing associative functions is harder than reaching consensus in faulty environments.

3 Preliminaries

We now show that as long as the distributed system remains connected in spite of the failure of m processors, it is possible for all of the non-faulty processors to construct V (the set of processor *ids*) and find the value of $|V|$. A message sent by one faulty processor can be delivered to all of the non-faulty processors by flooding – the sender sends messages on all of its links; a processor, on receiving such a message, if it is not the intended recipient, forwards it on all of its links if

```
procedure construct_m_graph;
{ executed by processor $u$ }
begin
    $V(u) \leftarrow \{u\} \cup \{v \mid v$ is a neighbor of $u\}$;
    $E(u) \leftarrow \{(u,v) \mid v$ is a neighbor of $u$ $\}$;
    mark $u$ of $V(u)$; { all others are unmarked }
    broadcast $V(u)$ and $E(u)$;
    while unmarked processors in $V(u) >$ $m$ loop
        wait for a message from $w$ containing
            $V(w)$ and $E(w)$;
        $V(u) \leftarrow V(u) \cup V(w)$;
        $E(u) \leftarrow E(u) \cup E(w)$;
        mark $w \in V(u)$;
    endloop;
{ $V(u) = V$ }
{ $G=(V(u),E(u))$ is the complete topology if $m = 1$ }
end;
```

Figure 1: Algorithm *construct_m_graph*

this is the first such message, and ignores the message if this message has already been received. Note that a message can be sent to all of the processors by flooding even if the transmission is point-to-point as long as the distributed system remains connected. Thus, any two non-faulty processors can communicate with each other.

Initially, a processor u initializes $V(u)$ and $E(u)$ using locally known information. It also *marks* $u \in V(u)$ to reflect the fact that complete information from u has been received. It then broadcasts $V(u)$ and $E(u)$. Since the distributed system remains connected in spite of the processor failures, each message sent by a processor will be eventually received by another processor if the sender and the receiver are non-faulty. When u receives $V(w)$ and $E(w)$ from w, it adds them to $V(u)$ and $E(u)$ respectively, and marks $w \in V(u)$. This step is repeated until at most m unmarked processors remain in $V(u)$. Algorithm *construct_m_graph* in Figure 1 contains the details.

A processor terminates the execution of algorithm *construct_m_graph* as soon as it receives adjacency lists from $|V| - m - 1$ processors. Since at most m processors are faulty, each non-faulty processor receives messages from at least $|V| - m - 1$ processors, marks at least m processors, and terminates algorithm *construct_m_graph* locally.

Lemma 1 *After processor u terminates algorithm* construct_m_graph, *$V(u)$ is the set of all of the (faulty and non-faulty) processors.*

Proof Assume for contradiction that $G=(V,E)$ represents the topology of the distributed system and $V(u) \neq V$. Let V'' be the set of processors that are unmarked by u at the end of algorithm *construct_m_graph* ($|V''| \leq m$). Thus, $V'' \subseteq V(u) \subset V$. Consider the resulting graph $G' = (V', E')$ after removing processors in V'' and the edges incident on them from G. Since G remains connected in spite of the failures of m processors, G' is a connected graph. Thus, there exists an edge (w, w') of G' such that $w \in V(u)$ is marked by u and $w' \in V - V(u)$. Since w is marked, $w' \in V(u)$, a contradiction.■

Theorem 1 *For $m=1$, algorithm* construct_m_graph *constructs $G=(V(u),E(u))$ locally at all processors in spite of the failure of a single processor where G is the complete network topology.*■

4 Computing Associative Functions

Protocols for a wide class of problems (distributed deadlock detection [2], constructing global states [13], minimum spanning tree [7], breadth first search [1], shortest paths [9], maximum flow [4, 11], maximum matching [14], sorting [15], median finding [12], etc.) use the following paradigm: (a) construct a directed spanning tree rooted at a *coordinator* and (b) decompose the protocol into several phases such that the following computation takes place in each phase: the value of a function of the private values (that might change from phase to phase) of each processor is computed and a certain decision is made which advances the computation. The computation of the function is carried out using a *convergecast* technique (also known as shout-echo [3]) to minimize the number of messages as follows: each leaf of the spanning tree computes the function based on its private value(s) and sends the result to its parent; an internal processor, after receiving the results from all of its children, computes the function using these results and its own private value(s), and sends the result to its parent. The coordinator computes the function using the results received from its children and its own private value(s). At the end of each phase, the coordinator either initiates a new phase or signals the end of the protocol. It is clear that any associative function can be computed in this manner.

Let us call the distributed protocols of the above type **af-based protocols** (based on computation of associative functions). Since there are exactly $|V|-1$ links in a spanning tree, each phase of an af-based protocol can be implemented using $O(|V|)$ messages only (assuming that the value of the associative function at each processor can be represented by a constant number of messages). A common theme in all of the above-mentioned protocols is the repeated computation of an associative function. Our main focus in this paper is on the problem of computing associative functions in spite of processor (processor) failures.

4.1 Fault-tolerant Associative Function Computation

We say that it is possible to compute an associative function of the private values of the processors in the presence of certain faulty processors if there exists a distributed protocol *fn* with the following properties:

1. Each processor must either terminate its part of the protocol *fn* after some arbitrary but finite number of local steps (said to be non-faulty) or take a failure step (said to be faulty).

2. The private value of each non-faulty processor must be used in the function computation.

3. The value of the function computed by each non-faulty processor must be the same.

We next define *failure detection* and show that computing an associative function is possible in a distributed system if and only if failure detection is possible.

4.2 Failure detection

The failure of a processor p can be detected if there exists a distributed protocol fd_p with the following properties:

1. Every processor starts its local algorithm of protocol fd_p (denoted by $fd_p(q)$ for processor q) either spontaneously or on receipt of a message from a neighbor requesting it to start $fd_p(q)$.

2. After a finite number of steps, each processor terminates its local algorithm and reaches a final state (said to be *non-faulty*), or it takes a failure step (said to be *faulty*).

3. Each non-faulty processor q decides that p is faulty only if q will not receive any message from p *after* q terminates $fd_p(q)$. Processor q decides that p is non-faulty only if some processor r (not necessarily different from q) receives a message from p before r terminates $fd_p(r)$.

4. All of the non-faulty processors agree on the same result. That is, all such processors either decide that p is faulty or p is non-faulty.

Processor failure detection is *possible* if there exist protocols fd_p for each $p \in$ V that satisfy the above conditions. Note that it is possible for a processor q to (locally) decide that p is faulty even after q receives a message from p.

4.3 Equivalence of failure detection and computing associative functions

Lemma 2 *If processor failure detection is possible, then associative functions can be computed in faulty distributed systems.*

Proof An associative function can be computed as follows: The processors broadcast messages requesting the private values of all of the processors, and in response, each processor sends its value by flooding. Now, each processor waits for $|V| - m - 1$ private values, and it has its own private value. Each processor p_i does the following:

Let $\{q_{i_1}, \ldots, q_{i_m}\}$ be the set of m processors from which p_i has not received private values. For each q_{i_j}, processor p_i starts the failure detection protocol $fd_{q_{i_j}}$, and at the same p_i waits for the private value of q_i. When the private value of q_i is received by p_i, processor p_i stores it locally. If $fd_{q_{i_j}}$ terminates and reports q_{i_j} to be non-faulty, then p_i restarts $fd_{q_{i_j}}$. Processor p_i repeats this until it receives the private value of q_{i_j} or $fd_{q_{i_j}}$ declares q_{i_j} to be faulty for $j = 1, \ldots, m$. Processor p_i uses its private value and all of the private values received and computes the associative function. To ensure that the value of the function is the same, the leaders run a consensus protocol and broadcast the result. Since failure detection is possible, the consensus problem can be solved by using any protocol that solves the Byzantine generals problem.

Note that the private value of each non-faulty processor is used in computing the function. Clearly the protocol satisfies the three rules of computing an associative function.■

Lemma 3 *If associative functions can be computed in a faulty distributed system, then processor failure detection is possible.*

Proof To detect the failure of say p, the set theoretic function set union is computed distributively in the presence of a faulty processor. (Note that set union is an associative function.) The private value of a processor is its identifier. If the identifier p belongs to the union, then all of the non-faulty processors infer that p is non-faulty. If not, they infer that p must have failed. If the identifier p is included in the result of the function union, then p must have sent a message (as the private value of p is known only to itself) and it must have been received by a processor r before terminating $fd_p(r)$. If p does not belong to the result of union, then from the rule that every non-faulty processor's private value must be used in computing the function, it is clear that p is faulty.■

Theorem 2 *In any system model, an associative function can be computed in spite of processor failures if and only if processor failure detection is possible.*■

Intuitively, in the process of computing an associative function, either the private value of a processor p must be used in computing the associative function or p must be declared to be faulty. Thus, a non-faulty processor cannot indefinitely delay the computation of an associative function.

5 Failure Detection

Since the failure detection problem solves the consensus problem, we consider only the following four (minimal) cases in which consensus is possible in spite of processor failures:

1. Synchronous communication and atomic receive/send (and broadcast transmission for multiple processor failures).
2. Synchronous processors and synchronous communication.
3. Synchronous processors and synchronous message order.
4. Broadcast transmission and synchronous message order.

For the first three cases, we show that it is possible to cope with processor failures in the context of computing associative functions. In §5.2, we show that it is impossible to cope with the failure of one processor in computing associative functions in the fourth case.

5.1 Protocols for Failure Detection

1. In a system where the communication is synchronous and receive/send is atomic, the protocol fd_q is as follows: A **timeout** strategy can be used by a processor p to infer if its neighbor q has failed - processor p sends a message to q and waits for an acknowledgement from q. Since the communication is Δ-synchronous, the message will be received by q within Δ real time steps, and since receive/send is atomic, the message will be received and processed by q and an acknowledgement can be sent in one atomic step. Thus, p waits for 2Δ real time steps for an acknowledgement. Meanwhile, if p receives an acknowledgement, then p locally decides that q has not failed, and if p does not receive an acknowledgement, then p decides that q has failed. In either case, p sends its local decision to all of the other processors by a broadcast. The above procedure is executed by every neighbor of q. A processor that is not a neighbor of q waits for a message from one of the neighbors of q with a local decision and chooses the first decision it receives. Thus, each processor has a local decision. Now all of the processors run a consensus protocol (for example, the protocol of [5]) to agree on the common value for the result of fd_q.
2. If the processors and the communication are synchronous, then the traditional timeout strategy can be used by a processor p to detect the failure of its neighbor q. The protocol fd_q is similar to the protocol presented in case 1 and the details are omitted.
3. In the third case, processors and message order are synchronous. Each processor sends an *active* message to all of its neighbors, one by one. Also, at every other step, it sends a message to itself. These two activities are external to the failure detection process and are part of the "normal" behavior of the processor. Since processors are Φ-synchronous, if one processor takes $\Phi + 1$ steps, then all of the other non-faulty processors must take at least one step in the mean time. Thus, each processor waits until it receives $|V|(\Phi + 1)$ messages sent by itself. In the mean time, if it does not receive an *active* message from a neighbor q, then by the synchrony of the processors and the message order, it is clear that q has failed. Thus, the failure of a processor can be detected by its neighbor(s). The rest of the steps of fd_q are similar to case 1.

5.2 Impossibility of Failure Detection

We show that it is impossible to detect processor failures if the transmission mechanism is broadcast and message order is synchronous, and from the equivalence of failure detection and computing associative functions, the result follows. First, we assume that a fault-detection protocol fd_q to detect the failure of q exists. We then show that there exists an execution of the protocol in which no processor terminates after a finite number of steps.

For each processor p, let Σ_p be the set of possible internal states for p. Among the states in Σ_p, there are some states called the ***initial states*** and there are some states called the ***final states.*** Processor p starts from an initial state and, if it ends in a final state, it (processor p) terminates. The communication system is represented by a message buffer which is implemented as an array of queues, one queue for each processor. The following operations are possible on the message buffer:

1. send(q, m) - append m to the end of the queue of processor q.

2. receive(p) - find message m at the head of the queue for p, delete it from the queue of p and deliver m to p (a null message is returned if the queue is empty).

3. broadcast(m) - append a copy of message m to the end of the queues of all of the processors.

Since messages (transmitted by a send or a broadcast operation) are immediately appended to the message queues of the appropriate processors, synchronous message order is preserved by the above operations. As the communication is asynchronous, the message buffer can return a null message a finite number of times in response to a receive(p) even though the queue of p is not empty.

The local algorithm of a processor p is represented by a transition function δ_p: $\Sigma_p \times M \rightarrow \Sigma_p \times M$ where M represents a set of pairs of the form (r, m) where r is a processor *id* and m is a message. The local action of a processor p in a step (not necessarily atomic) is as follows: it performs a receive(p) operation, receives message m from the message buffer (communication medium), applies the transition function, changes its state, and sends a (possibly empty set of) message(s) or it broadcasts a message. This action is denoted by an *event* $e = (p, m)$. Processor failure is indicated by a failure step $(p, \star)$ where $\star$ denotes the death of processor p. When a processor takes a failure step, all of its subsequent steps are failure steps.

A *configuration* consists of the internal states of the processors and the contents of the message buffer. An *initial configuration* is a configuration in which all of the processors are in one of their respective initial states and the message buffer is empty. A ***final configuration*** is a configuration in which each processor either is in one of its final states or it took a failure step.

A step of a distributed system takes the system from one configuration to another configuration. An event $e = (p, m)$ is said to be ***applicable*** to a configuration if $m = \phi$ or m is the message at the head of the message queue of p. Let C be a configuration and let e be an event applicable to C. The configuration that results from C after the event e takes place is denoted by e(C). Let σ be a sequence e_1, e_2, $\ldots e_k$ of events. We say that σ is applicable to the configuration C if e_1 is applicable to C, e_2 is applicable to e_1(C), ..., and e_k is applicable to $e_{k-1}(\ldots e_1(C)\ldots)$. Any σ that is applicable to a configuration C is a ***run.*** A run σ is said to be a finite run if σ consists of a finite number of events. Let σ(C) denote the resulting configuration $e_k(e_{k-1}(\ldots e_1(C)\ldots))$. A configuration C$'$ is said to be ***reachable from*** C if there exists a σ that is applicable to C such that $C' = \sigma(C)$. Any configuration that is reachable from an initial configuration is said to be a ***reachable configuration.***

We say that it is possible to reach the decision f (decision n) from a configuration C if there exists a finite run σ that is applicable to C such that σ(C) is a final configuration and each processor that reaches a final state in σ(C) decides that q has failed (not failed for decision n). A configuration is ***f-valent*** (***n-valent***) if the only possible decision that can be reached from that configuration is f (decision n). Configuration C is said to be a ***bivalent configuration*** if it is possible to reach either of the decisions f and n by finite runs starting from C. A reachable bivalent configuration C is said to be a ***safe bivalent configuration*** with respect to q if there exists an initial configuration I and a run σ such that $C = \sigma(I)$ and there is no event in σ that results in a message sending by q.

Lemma 4 *There exists a safe bivalent initial configuration with respect to q for any processor q.*■

We now show that there exists an an infinite run for any protocol that solves the failure detection problem.

Lemma 5 *Let C be a safe bivalent configuration and let e=(p,m) be an event applicable to C where $p \neq q$. Let D be the set of all configurations reachable from C by finite runs in which p and q do not take any steps. Let E={e(d) | d ∈ D}. E contains a safe bivalent configuration.*

Proof If e is applicable to C, then e is applicable to each $d \in$ D also because the processors are asynchronous. Assume that all of the configurations in E are univalent configurations. Consider a configuration c in E. Since each configuration in E is univalent, either c must be an f-valent or an n-valent configuration. In both cases, we arrive at a contradiction.

Assume first that c is an f-valent configuration. Let $e' = (q, m)$ be an event such that e' results in q broadcasting a message, and let $e'(c) = c_2$ be the resulting configuration. There exists a final configuration c_3 reachable from c_2 such that the decision reached is n (to see that, note that q broadcasts a message when e' is applied and all of the processors will eventually receive the message broadcast by q). This implies that there exists a run in which it is possible to reach the decision n from c. Thus, c is not f-valent, a contradiction.

A similar proof applies if c is n-valent. To see that the bivalent configuration in E is safe, recall that we started with a safe bivalent configuration, and in arriving at a configuration in E, q does not take any steps.■

Intuitively, the absence of a message from q can be due to the failure of q or due to the asynchrony of q, and it is impossible for the other processors to distinguish between the two cases. To prove the impossibility result, we construct an infinite run in which no processor can reach a final state after a finite number of local events. The run is constructed using the result of Lemma 5. Keep a queue of processors and let the processor p ($p \neq q$) at the head of the queue take a step using the message at the head of the queue if that processor taking that step results in a bivalent configuration. Otherwise, the result of Lemma 5 is used to obtain a bivalent configuration in which the last event is on the processor p. After applying the event $e = (p, m)$, put the processor p at the tail of the queue. Thus, all processors other than q take infinite number of steps and no processor terminates after a finite number of steps.

Theorem 3 *Processor failures cannot be detected in a system with broadcast transmission and synchronous message order.*■

Theorem 4 *It is impossible to compute an associative function with a possibly faulty processor if the only types of synchrony in the system are broadcast transmission and synchronous message order.*■

6 Discussion

While the results of the previous section have proven that the case where both processors and communication are synchronous is the only realistic case when processor failures can be tolerated at reasonable costs, reinforcing the intuition, note that this is not so in solving all problems distributively. If no more than a single processor can fail during the execution of a protocol, then we can solve graph-theoretical problems in *any system model.* Intuitively, graph-theoretic problems can be solved since the information about the set of links incident on a processor and the *id* of the processor are available to the other processors even if p fails.

This result does not contradict the impossibility result of Fischer et al. [6] since they consider the nontrivial consensus problem while the above protocol results in a trivial consensus protocol (where all of the processors decide on the value say 0). For the same reason, the impossibility result of Moran and Wolfstahl [8] does not contradict the possibility of solving any graph-theoretic problem.

Theorem 5 *If the processors and communication are asynchronous, then topological problems cannot be solved in the presence of two or more faulty processors.*

Proof Assume for contradiction that there exists a protocol Π_1 that solves topological problems in the presence of two faulty processors when processors and communication are asynchronous. Assume that Π_1 constructs a spanning tree. We now present a solution to the consensus problem in spite of processor failures in completely connected asynchronous distributed systems.

Let G′=(V′,E′) represent a completely connected network where each processor knows the identities of its neighbors. Since G′ is completely connected, V′ is initially known to each processor $u \in$ V′. All of the processors run Π_1 to construct a spanning tree. The input to Π_1 at processor u consists of its own identity (u) and the identities of its neighbors (and the links) only. Protocol Π_1 does not know that G′ is completely connected. After a processor completes Π_1, the spanning tree is available locally at u. Each processor u decides on a value 1 for the consensus problem if link (v_1, v_2) belongs to the spanning tree and u decides on value 0 otherwise, where v_1 and v_2 are the two smallest processor indentities of V′. There are infinitely many runs of Π_1 in which the processors decide on value 1 and infinitely many runs in which the processors decide on value 0, and all of the processors decide on the same value for each execution of Π_1. Thus, there is a solution to the consensus problem contradicting the impossibility result of [6].■

Intuitively, in asynchronous systems, information about a communication link between two processors can be hidden from all of the other processors if the two processors are slow/faulty, and it is impossible to distinguish between a slow and a faulty processor.

Theorem 6 *If the identities of the neighbors are unknown in asynchronous systems, then constructing V at each processor is impossible in the presence of single faulty processor, while V can be constructed in spite of multiple processor failures not leading to a partition if the identities of the neighbors are known.*

Proof The proof of the first part of this theorem is similar to the proof of Theorem 5 and is omitted.

The second part of the theorem follows from the proof of correctness of algorithm *construct_m_graph* of Section 3.■

7 Conclusions

Computing an associative function is a commonly used paradigm in distributed systems. Thus, the main focus of this paper is on computing associative functions in spite of processor failures. We have shown that associative functions can be computed in spite of processor failures if and only if processor failure detection is possible. Various models of distributed systems are considered in the context of computing associative functions. In each case in which it is possible to compute associative functions (in spite of processor failures), we have presented a protocol. In one case, we have proved that it is impossible to cope with the failure of one processor. The only realistic case in which it is possible to cope with processor failures is when the processor and the communication links are synchronous.

References

[1] Awerbuch, B., and Gallager, R. Distributed BFS algorithms. In *Proceedings of the Twenty Sixth Annual Symposium on Foundations of Computer Science* (1985), pp. 250–256.

[2] Chandy, K., Misra, J., and Haas. Distributed deadlock detection. *ACM Trans. Comput. Syst. 1*, 2 (1983), 144–156.

[3] Chang, E. Echo algorithms: Depth parallel operations on general graphs. *IEEE Trans. Softw. Eng. SE-8*, 4 (1982), 391–401.

[4] Cheung, T. Graph traversal techniques and the maximum flow problem in distributed computation. *IEEE Trans. Softw. Eng. SE-9*, 4 (1983), 504–512.

[5] Dolev, D., Dwork, C., and Stockmeyer, L. On the minimal synchronization needed for distributed consensus. *J. ACM 34*, 1 (1987), 77–97.

[6] Fischer, M., Lynch, N., and Paterson, M. Impossibility of distributed consensus with one faulty process. *J. ACM 32*, 2 (1985), 374–382.

[7] Gallager, R., Humblet, P., and Spira, P. A distributed algorithm for minimum weight spanning trees. *ACM Trans. Program. Lang. Syst. 5*, 1 (1983), 66–77.

[8] Moran, S., and Wolfstahl, Y. Extended impossibility results for asynchronous complete networks. *Inf. Process. Lett. 26*, 3 (1987).

[9] Ramarao, K., and Venkatesan, S. On finding and updating shortest paths distributively. *Journal of Algorithms 13*, 2 (1992), 235–257.

[10] Schneider, F. Byzantine generals in action: Implementing fail-stop processors. *ACM Trans. Comput. Syst. 2*, 2 (1984), 145–154.

[11] Segall, A. Decentralized maximum-flow protocols. *Networks 12* (1982), 213–230.

[12] Shrira, L., Francez, N., and Rodeh, M. Distributed k-selection: From a sequential to a distributed algorithm. In *Proceedings of the Second Symposium on Principles of Distributed Computing* (1983), pp. 143–153.

[13] Venkatesan, S. Message-optimal incremental snapshots. In *Proceedings of the Ninth International Conference on Distributed Computing Systems* (1989), IEEE, pp. 53–60.

[14] Wu, C. An efficient distributed algorithm for maximum matching in general graphs. Tech. rep., University of Illinois, Urbana, 1987.

[15] Zaks, S. Optimal distributed algorithms for sorting and ranking. *IEEE Trans. Comput. C-34*, 4 (1985), 376–379.