Communication Adaptive Self-Stabilizing Group Membership Service

(Extended Abstract)

Shlomi Dolev[1⋆] and Elad Schiller[1]

Department of Computer Science, Ben-Gurion University — Israel
{dolev,schiller}@cs.bgu.ac.il

Abstract. This paper presents the first algorithm for implementing self-stabilizing group communication services in an asynchronous system. Our algorithm converges rapidly to a legal behavior and is *communication adaptive*. Namely, the communication volume is high when the system recovers from the occurrence of faults and is low once a legal state is reached. The communication adaptability is achieved by a new technique that combines transient fault detectors.

1 Introduction

Group communication services are becoming widely accepted as useful building blocks for the construction of fault-tolerant distributed systems and communication networks. Designing robust distributed systems is one of the most important goals in the research of distributed computing. One way to simplify the design and programming process of a distributed system is to design a useful set of programming high-level primitives that forms a robust set of tools. A group communication system enables processors that share a collective interest, to identify themselves as a single logical communication endpoint. Each such endpoint is called a *group*, and is uniquely identified. A processor may become a *member* of or departure a group, by issuing a join/leave request to the *group membership service*. The membership service reports membership changes to the members, by delivering *view*s. A *view* of a group includes a set of members and a unique identifier. A processor may send a message to a group, using a group *multicast service*.

A very important property in the implementation of the primitives of group communication services is its fault tolerance and robustness. It is assumed that processors leave and join the group either voluntarily or due to crashes or recoveries. The distributed algorithms that implement these services assume a particular set of possible failures, such as crash failures, link failures or messages loss. The implementing algorithms should provide the specified services in spite of the occurrence of these failures. The correctness of the implementing algorithms is proved by assuming a predefined initial state and considering every possible execution that involves the assumed possible set of failures.

⋆ Dolev's work was supported by BGU seed grant.

A.K. Datta and T. Herman (Eds.): WSS 2001, LNCS 2194, pp. 82–97, 2001.

The abstraction that limits the possible set of failures is convenient for establishing correctness of the algorithm, but it can at the same time be too restrictive. Group communication service is a long-lived on-line task and hence it is very hard to predict in advance the exact set of faults that may occur. Therefore, it may be the case that due to the occurrence of an unexpected fault, the system reaches a state that is not attainable from the initial state by the predefined transitions (including the predefined fault occurrences). Self-stabilizing systems [11, 14] can be started in any arbitrary state and exhibit the desired behavior after a convergence period. Thus, self-stabilizing group communication services can automatically recover following the occurrence of such unexpected faults.

Related Work: The Isis system [8] is the first implementation of group communication services that triggered researchers and developers to further examine such services. Cristian [9] formalized a definition of group communication for synchronous systems. Group communication services were implemented with different guarantees for reliability and message delivery order. For example, Isis [8], Transis [17], Totem [23], Newtop [19], Relacs [7], Horus [25] and Ensemble [26]. None of the above implementation is self-stabilizing. A specification that guarantees performance once the system stabilizes to satisfy certain properties is presented in [20]. This is a consequence of existing impossibility results for requirements that hold in all possible executions e.g., [10]. Still it is assumed in [20] that the system is started in a certain global state, and the transitions are from a predefined set of transitions — thus the specification and algorithm presented in [20] are not designed for self-stabilizing systems.

A different approach (part of which is randomized) is used in [27]. Every processor periodically transmits a list of the processors that it can directly communicate with. A processor is consider "up" and connected as long as it can successfully transmit a "fresh" time-stamp; otherwise it will be eventually discarded from the system. The algorithm presented in [27] may be a base for a self-stabilizing algorithm, if for example, each processor has access to a local pulse generator, such that the maximum drift between the pulse generators is negligible.

Congress [2] is an elegant protocol for registration of membership information at (hierarchically organized) servers. Hierarchy of servers improves scalability. Users send a message to servers with join or leave requests. The servers maintain the membership information. The design fits wide area network using virtual links to define neighboring relation.

Moshe [21] is a group membership service implementation, that considers an abstract network service (such as Congress [2]). The network service monitors the up and connected members of every group and delivers multicast messages to the members of a group. The common cases of membership changes (joins/leaves) are considered in order to achieve scalability. The group membership algorithm of Moshe uses unbounded counters.

A self-stabilizing group membership service for synchronous systems is considered in [3]. A common periodic signal initiates a broadcast of local topology of every processor. Every processor uses the local topologies in order to compute

the connected component to which it belongs. Unbounded signal numbers are used and changes in the group are discovered following a common signal.

Our Contribution: this paper presents the first algorithm for implementing a self-stabilizing group membership service in asynchronous systems. We assume that processors eventually know the set of non-crashed processors with which they can communicate directly. We show that once every processor knows the correct set, the membership task is achieved within the order of the diameter of the communication graph. Moreover, the activity of the processors is according to the required group membership service. Our algorithm converges rapidly to a legal behavior and is *communication adaptive*. Namely, the communication volume is high when the system recovers from the occurrence of faults and is low once a legal state is reached. The communication adaptability is achieved by a new technique that combines transient fault detectors. Furthermore, randomized techniques can be used to dramatically reduce the communication complexity of the deterministic transient failure detector.

Our group membership service can be extended to implement different levels of broadcast services, such as single-source FIFO; totally ordered; and causally ordered.

The rest of the paper is organized as follows. The system settings appear in Section 2. Our algorithms for implementing a self-stabilizing group membership service appear in Section 3. Concluding remarks are in Section 4. The proofs are omitted from this extended abstract, more details can be found in [18].

2 The System

The *distributed system* consists of a set of P communicating entities. We call each entity a *processor*, and assume that $1 \leq |P| = n \leq N$, where N is an upper bound on the number of processors. The processors may represent a network of real physical CPU, or correspond to an abstract entity like a process or thread in a timesharing system. Processors are connected by communication links through which they communicate by exchanging messages. $neighbors_i$ is the set of processors that processor p_i can directly communicate with. The communication link may represent a (real physical) communication channel device attached to the processor, a virtual link, or any inter-process communication facility (e.g., UDP, or TCP connections).

It is convenient to represent a distributed system by a communication graph $G = (V, E)$, where each node represents a processor and each edge represents a communication link. Let $p_i, p_j \in P$, $p_j \in neighbors_i$ iff $(p_i, p_j) \in E$.

The system is asynchronous. We assume however that processors eventually identify the crashed/non-crashed status of their attached links and neighbors. We sometime use the term *time-out* in the code of the processors for a repeated action of the processors. In fact, a zero time-out period will result in the desired behavior as well. The time-out period may only reduce the number of messages sent when processors have access to a time device.

A state machine models each processor. The communication links are modeled by two anti-directed FIFO queues. We use a (randomized) self-stabilizing

data-link algorithm on every link [1]. The existence of the self-stabilizing data-link algorithm ensures that when a message is sent it arrives to its destination before the next message is sent. Thus, input communication buffers or communication registers (when buffers contain at most one message and when the content of an arriving message replaces the previous content of the buffer) can be assumed whenever it is convenient instead of message passing.

The system configuration is a vector of the states of the processors and the values of the queues (the messages in the queues).

A *communication operation* is an operation in which a message is sent or a message is received. We also allow a processor to send the same message to every one of its neighbors in a single communication operation. A step of a processor consists of internal computations that are followed by a single communication operation. A system *execution* is an alternating sequence of configurations and (atomic) steps.

Processors may crash and recover during the execution. The neighbors of a crashed processor eventually identify the fact that it is crashed.

The program of a processor used here consists of a do-forever loop that includes communication step with every neighboring processor. Let R be an execution and let A be a connected component of the system such that no processor in A is crashed during R. The first *asynchronous cycle* of A in R is the minimal prefix of R such that each processor p_i in A communicates with every of its neighbors: At least one message m_j is sent by p_i to every neighbor p_j, such that p_j receives m_j during the asynchronous cycle.

The number of messages sent over a particular communication link during an asynchronous cycle is a function of the number of loop iterations the attached processors execute during this asynchronous cycle (note that a processor may execute any number of iterations before another processor completes a single loop iteration). Thus we consider a special execution to measure the communication complexity of an algorithm. A *very fair execution* is an execution in which every processor executes exactly a single iteration of its do-forever loop in every asynchronous cycle. The *communication complexity* is the total number of bits communicated over the communication links in a single asynchronous cycle of a *very fair execution*.

The set of *legal executions* includes all the executions that exhibit the desired behavior (input output relation) of the system for a task τ. For example, if τ is the mutual exclusion task, then at most one processor is executing the critical section in any configuration of a legal execution.

A *safe configuration* of the system is a configuration from which only legal executions, with respect to τ, start.

In this paper, the requirements are related to the *eventual* behavior of the system when the execution fulfills certain properties (unlike the requirements discussed in [10] see also [24]). We require that a self-stabilizing algorithm for group communication service will reach a safe configuration within a certain number of asynchronous cycles in any execution (that starts in an arbitrary con-

figuration) such that each processor p_i has a *fixed set of non crashed neighbors* during the execution[1].

We allow simultaneous existence of several groups. We do not consider however, interaction between the groups. Therefore, we choose a specific group g for describing the membership service. A boolean variable $member_i$ (logically) represents the intention of p_i to be included in g. A partition of the network may cause a "partition" of g as well. Therefore, we associate the set of *legal executions* for the group membership task, with the processors of a (fixed) connected component A, and include execution R such that the following properties hold:

1. If the value of $member_i = true$ ($member_i = false$) is fixed during R then there exists a suffix of R, in which p_i appears (does not appear, respectively) in all the views of group g in the connected component A.
2. If the value of $member_i$ of every processor p_i of group g in the connected component A is fixed during R then there exists a suffix, in which all the views of group g in connected component A are identical, the views have the same list of members and the *same view identifier*.

We note that the length of the prefix of R before the suffix mentioned in the above requirement is achieved by our algorithms is $\Theta(d)$ (which is the fastest possible).

The communication of a self-stabilizing algorithm is *adaptive* if the maximal communication complexity after reaching a safe configuration is smaller than the maximal communication complexity before reaching a safe configuration.

3 Self-Stabilizing Group Membership Service

In this section we present the first communication adaptive self-stabilizing algorithm for the membership service. Roughly speaking, a spanning tree of the system is constructed. This tree is used to execute the membership management tasks. The root of the tree is responsible for the management of the membership requests, and establishing new views. Several transient fault detectors monitor the consistency of the tree and the membership information. The transient fault detectors give fast indication on the occurrence of transient faults. Once a fault is detected the system changes state to a safe configuration executing a *propagation of information with feedback* (*PIF*) procedure [28,13], for several times (choosing random identifiers for these executions to ensure eventual stabilization).

The update algorithm informs each processor with the nodes in its connected component. The update algorithm stabilizes fast, as it takes $\Theta(d)$ asynchronous cycles before reaching a safe configuration. We use d to denote the actual diameter of the connected component. Unfortunately, the communication complexity of the update algorithm is $O(|E|n \log n)$ before *and after* a safe configuration is reached. In this section we present an algorithm that reduces the communication complexity to $O(|E| \log n + n^2 \log n) = O(n^2 \log n)$ once a safe configuration is reached.

[1] We note that we do not consider the time required to identify the status of the links and neighbors.

A transient fault detector is composed with the update algorithm to achieve the communication adaptability property (see [4,6,5] for definitions of transient fault detectors). The transient fault detector signals every processor whether or not it needs to activate the update algorithm. Our transient fault detector itself is obtained by using a new technique for composing transient fault detectors.

Roughly speaking, whenever a processor detects, by use of the transient fault detector, that the update algorithm is not in a safe configuration, the processor signals the processors in the system to start the activity of the update. The processor stops signaling the other processors to operate the update algorithm when it receives an indication that a safe configuration is reached.

3.1 Self-Stabilizing Update

We use the self-stabilizing update algorithm of [12,15]. We now sketch the main ideas used by the update algorithm. We start with the data structure used by a processor. Each processor has a list of no more than N tuples $\langle id, dis, parent\rangle$. When the update algorithm stabilizes it holds that the list of a processor p_i contains n tuples, exactly one tuple $\langle j, dis, k\rangle$ for each processor p_j that is in the same connected component with p_i. The value of dis is the number of edges in a shortest path from p_i to p_j and p_k is a neighbor of p_i that is in a shortest path to p_j. Thus, when the algorithm stabilizes every processor knows the identities of the other processors in its connected component.

The processors that execute the update algorithm repeatedly receive all the tuples from the tables of their neighbors and use the value received to calculate a new table (note that the current table is not used in calculating the new table). Every time a processor p_i finishes receiving the tuples of its neighbors it acts as follows: Let $\mathcal{TU}_i$ be the set of all tuples that a processor p_i reads from its neighbors. p_i adds 1 to the dis field of every tuple in $\mathcal{TU}_i$. p_i adds a tuple $\langle i, 0, nil\rangle$ to $\mathcal{TU}_i$. If there are several tuples with the same id in the resulting $\mathcal{TU}_i$ then p_i removes every such tuple except a single tuple among these tuple, a tuple with the minimal dis value. Finally, p_i removes every tuple $\langle id, dis, parent\rangle$ such that there exists a positive $z < dis$ and there is no tuple with $dis = z$ in $\mathcal{TU}_i$. The resulting set in $\mathcal{TU}_i$ is the new table of p_i.

3.2 Transient Fault Detectors for Reducing Communication Overhead

The communication complexity of the update algorithm is $O(|E|n \log n)$. Note that a naive approach for designing a transient fault detector is to repeatedly send $\mathcal{TU}_i$ to every neighbor. A fault will be detected whenever there should be a change in the value of $\mathcal{TU}_i$ (according to the update algorithm) when a message with $\mathcal{TU}_j$ arrives from a neighboring processor p_j. This approach results in communication complexity that is identical to the communication complexity of the update algorithm.

In this section we present a fault detector that reduces the communication complexity of our algorithm when the algorithm stabilizes (reaches a safe configuration). The communication complexity of the algorithm when a fault detector is used is $O(n^2 \log n)$.

The update algorithm informs each processor with the nodes in its connected component. The task of the transient fault detector is to detect a fault whenever there exists at least one processor that does not know the set of processors in its connected component.

We present a new scheme for combining fault detectors that results in low communication complexity. In order to reduce the communication complexity, we combine two transient fault detectors. The first one communicates short messages over all the links of the system and ensures that there is a marked rooted spanning tree. The short messages consist of the identifier of the common leader and the distance of the processor from this leader. The second transient fault detector assumes the existence of a spanning tree and communicates larger messages over the links of this tree. In fact, these messages consist of the description of the rooted spanning tree.

Transient Fault Detector for the Existence of a Tree Rooted at a Leader: The code for the first part of the transient fault detector appears in Figure 1. In the code we use the input $\langle leader_i, dis_i, parent_i \rangle$ which is defined by the output of the update algorithm. Let $\langle l, d, p \rangle$ be the tuple in $\mathcal{TU}_i$, such that l is the maximal value among the values of the *leader* variables in $\mathcal{TU}_i$. The value of $\langle leader_i, dis_i, parent_i \rangle$ is assigned by the values of $\langle l, d, p \rangle$. A change in the value of $\langle leader_i, dis_i, parent_i \rangle$ as well as in the $neighbors_i$ set triggers fault detection.

Lines 1 and 1a of the code ensure that the information for detection of a fault is sent from every processor to its neighbors once every timeout period. Line 1b ensures that the processor for which $leader_i = i$ has the value 0 in its dis variable and the value nil in the $parent_i$ variable. Line 2a ensures that all the processors have the same value in their leader variable and the distance of the parent of each (non-leader) processor p_i is one less than the distance of p_i from the leader.

Input: $\langle leader_i, dis_i, parent_i \rangle$ *(* updated by lower level *)*

1. **Upon timeout:**
   (a) for each $j \in neighbors_i$ send $\langle leader_i, dis_i, parent_i \rangle$.
   (b) if $leader_i = i$ and ($dis_i \neq 0$ or $parent_i \neq nil$) then fault is detected.
2. **Upon receiving** $\langle l, d, p \rangle$ **from** p_j
   (a) if ($l \neq leader_i$) or (($j = parent_i$) and ($dis_i \neq d + 1$)) then fault detected.

Fig. 1. Transient Fault Detector of p_i, for the Existence of a Tree Rooted at a Leader.

Define the directed graph $\mathcal{T} = (V, E)$ as follows: each node of the graph V represents a processor in the system (and vice versa). There exists a directed edge $(i, j) \in E$ if and only if the value of the parent field of the processor p_i, in the tuple of $\mathcal{TU}_i$ with the maximal id, is p_j.

Definition 1. *A directed graph is an in-tree if the undirected underlying graph is a tree and if every edge of the tree is directed towards a common root.*

To prove the correctness we show that if no processor detects faults during an asynchronous cycle, then $\mathcal{T}$ is an in-tree rooted at the common leader (the processor with the maximal identifier).

The Tree Update Algorithm: Before we continue with the transient fault detector, let us add a mechanism to distribute the description of $\mathcal{T}$ to every processor in the system. We augment each processor p_i with a variable $\mathcal{T}_i$ that should contain the description of $\mathcal{T}$.

Let $\mathcal{T}_i(p_j)$ be the component of $\mathcal{T}_i$ that is connected to p_i when the link from p_i to p_j is removed from $\mathcal{T}_i$. p_i repeatedly sends $\mathcal{T}_i(p_j)$ to every processor $p_j \in (\{parent_i\} \cup children_i)$. $parent_i$ is defined by the value p_j of the tuple $\langle l, d, p_j \rangle$ in $\mathcal{TU}_i$ such that $l = leader_i$. The $children_i$ set includes every neighbor p_j from which the last table $\mathcal{TU}_j$ received, includes a tuple $\langle l, d, p_i \rangle$ where $l = leader_i$. p_i repeatedly computes $\mathcal{T}_i$ using the last values of $\mathcal{T}_j(p_i)$ received from every processor $p_j \in (\{parent_i\} \cup children_i)$. p_i construct $\mathcal{T}_i$ from the above $\mathcal{T}_j(p_i)$ adding the links connecting itself to the processors in $(\{parent_i\} \cup children_i)$.

Input: $\mathcal{T}_i, parent_i, children_i$ *(* updated by lower level *)*

1. $consistent \leftarrow true$
2. if $\mathcal{T}_i$ does not encode a spanning in-tree then $consistent \leftarrow false$
3. if $children_i$ is different from the set of processors that are the children of p_i in $\mathcal{T}_i$ then $consistent \leftarrow false$
4. if $parent_i = nil$ and p_i is not the leader of $\mathcal{T}_i$ then $consistent \leftarrow false$
5. if $parent_i \neq nil$ and $parent_i$ is not the parent of p_i in $\mathcal{T}_i$ then $consistent \leftarrow false$
6. return $consistent$

Fig. 2. Consistency Test Function.

We now prove the correctness of the tree update algorithm. In the proof we consider an execution that starts in a *safe configuration* of the update algorithm and prove correctness of the tree update in such executions. A *safe configuration* of the update algorithm is a configuration in which the values of the tuples of all the processors are correct (and therefore are not changed in any execution that starts in such a safe configuration).

In the lemma we use the term *height* of a processor p_i in an in-tree for the maximal number of edges in a path from a leaf in the tree to p_i, such that the path does not include the root of the tree.

Lemma 1. *Consider any execution R of the tree update algorithm that starts in a safe configuration of the update algorithm and consists of at least $l + 1$ asynchronous cycles. Let p_j be a processor such that $\mathcal{T}(p_j)$ is a sub-in-tree of $\mathcal{T}$ (the in-tree defined by the update algorithm) that is rooted at p_j, and the height of $\mathcal{T}(p_j)$ is at most l. Let $\mathcal{T}_j(p_j)$ be the description of the tree rooted at p_j in the variable $\mathcal{T}_j$ of p_j. It holds that $\mathcal{T}(p_j) = \mathcal{T}_j(p_j)$ in the last configuration of R.*

A configuration, c, is safe with relation to the tree update algorithm iff c is safe for the update algorithm and for every processor p_i $\mathcal{T}_i = \mathcal{T}$. Moreover, in any execution that starts in c the value of $\mathcal{T}_i$ is not changed (this last requirement implies, in fact, that any message in transit from p_i to p_j contains $\mathcal{T}_i(p_j)$ that is the portion of $\mathcal{T}$ connected to p_i when the link from p_i to p_j is removed).

Corollary 1. *The tree update algorithm reaches a safe configuration following the first $O(d)$ asynchronous cycles and its communication complexity is $O(n^2 \log N)$.*

Transient Fault Detector for Correct Description of the Tree: The second transient fault detector assumes the existence of a rooted spanning tree $\mathcal{T}$ that is defined by the child parent relation and ensures that every processor p_i has the description of $\mathcal{T}$ in $\mathcal{T}_i$. Thus, ensures that every processor knows the set of processors in its connected component.

Let us first describe the consistency test function in Figure 2 that is used by our transient fault detector. In the code we use the input $\mathcal{T}_i, parent_i, children_i$ which is defined by the output of the tree update algorithm. The consistency test function uses a boolean variable *consistent*. First p_i assigns true to the *consistent* variable (line 1 of Figure 2). In line 2, p_i checks $\mathcal{T}_i$ to be a spanning *in-tree* — a directed tree for which every edge is directed towards a common root. Lines 3, 4 and 5 test the child parent relations of p_i (according to the update algorithm) to be correct in $\mathcal{T}_i$. The function returns the final value of *consistent*.

The transient fault detector is presented in Figure 3. The fault detector will ensure that all local values of $\mathcal{T}$ are identical and that every processor local tree neighborhood appears in $\mathcal{T}$. In the code we use the input $\mathcal{T}_i, parent_i, children_i$ which is defined by the output of the tree update algorithm (see the description of the code of Figure 2 for the values of the above inputs).

p_i repeatedly executes line 1a and 1b. In line 1a p_i sends $\mathcal{T}_i$ to its parent and children. p_i checks the consistency of $\mathcal{T}_i$ according to the consistency test described in Figure 2 and detects a fault accordingly. Whenever p_i receives $\mathcal{T}_j$ from p_j, p_i checks whether $\mathcal{T}_i = \mathcal{T}_j$ and detects a fault if this equation is not true (line 2a of the code).

Input: $\mathcal{T}_i, parent_i, children_i$ *(* updated by lower level *)*

1. **Upon timeout:**
   (a) for each $j \in \{parent_i\} \cup children_i$ send $\mathcal{T}_i$ to p_j.
   (b) if $\mathcal{T}_i$ is inconsistent then detected a fault.
2. **Upon receiving** $\mathcal{T}_j$ **from** p_j
   (a) if $\mathcal{T}_i \neq \mathcal{T}_j$ then detected a fault.

Fig. 3. Transient Fault Detector of p_i, for Correct Description of the Tree.

We prove the correctness of the second fault detector assuming that no fault is detected by the first transient fault detector.

Lastly, we combine both the fault detectors; the first fault detector messages are augmented with the second fault detector messages. (Note that the second fault detector sends messages only on tree links. The messages sent by the first fault detector on non-tree links are not augmented by a message of the second fault detector). We conclude the presentation and correctness proof of the transient fault detectors by the following corollary.

Corollary 2. *(1) The combined fault detector detects a fault during a single asynchronous cycle whenever there exists a processor p_i such that $\mathcal{T}_i$ does not consist of the processors in p_i's connected component, and (2) The communication complexity of the combined fault detector is $O(n^2 \log N)$.*

3.3 Lower Bound on the Communication Complexity

We now present a lower bound of $\Omega(n^2 \log(N/n-1))$ bits on the communication complexity. The lower bound is for any fault detector that detects a fault within a single asynchronous cycle (whenever a processor has an inconsistent knowledge on the set of processors in its connected component or view). Recall that group membership services notifies the application with the current view. To do so we consider an asynchronous cycle that starts with all processors sending to every one of their neighbors (where a processor can send *nil* messages in case no message should be sent to a neighbor), and the cycle terminates after all messages sent are received. We examine processors $p_1, p_2, \cdots p_n$ that are connected by a chain communication graph. Assume that n is even (a similar argument can be used for a chain with an odd number of processors).

Let $m_{k,k+1}$ ($m_{k+1,k}$) be the message sent from p_k to p_{k+1} (from p_{k+1} to p_k, respectively). We claim that the number of distinct combinations of $m_{k,k+1}, m_{k+1,k}$ must be at least $\Omega(n \log(N/n-1))$.

Let p_k be a processor in the chain and suppose that $k \leq n/2$. Fix a set of k distinct identifiers for the processors $p_1, p_2, \cdots, p_k$. We prove a lower bound by using the number of possible choices of different sets of $n-k$ distinct identifiers for the rest of the processors $p_{k+1}, p_{k+2}, \cdots, p_n$.

Let $\mathcal{X}_1$ and $\mathcal{X}_2$ be two such choices. Now we describe two different systems that differ in the way we assign identifiers to processors $p_{k+1}, p_{k+2}, \cdots, p_n$. The identifiers of the processors $p_{k+1}, p_{k+2}, \cdots, p_n$ in the first (second) system are the identifiers in $\mathcal{X}_1$ ($\mathcal{X}_2$, respectively). Clearly the communication over the edge connecting p_k to p_{k+1} must not be the same in the two systems above. Otherwise we may replace the two different portions of the two systems and no fault will be detected, while p_1 is not aware of the different set of processors in the system. The case of $kn/2$ is handled analogously fixing a set of k distinct identifiers for the processors $p_{n-k}, p_{n-k+1}, \cdots, p_n$. In both cases we conclude that the number of communication patterns needed are at least the number of choices of $n-k$ distinct identifiers for the processors $p_{k+1}, p_{k+2}, \cdots, p_n$, out of $N-k$ identifiers. $(N-k)!/((n-k)!((N-k)-(n-k))!) = (N-k)!/((n-k)!(N-n)!) = ((N-n+1)\cdots(N-k))/(n-k)! \geq ((N-n+1)/(n-k))^{n-k}$. We assume that $N \geq 2n$, thus we have that $(N-n+1)/(n-k) \geq 1$. By the assumption that $1 \leq k \leq n/2$, we have that $((N-n+1)/(n-k))^{n-k} \geq ((N-n)/n)^{n/2}$. Therefore,

for the communication between $m_{k,k+1}, m_{k+1,k}$, at least $\Omega(n \log(N/n-1))$ bits are needed. The communication complexity is a measure that considers all the links and therefore is $\Omega(n^2 \log(N/n-1))$ bits.

3.4 Group Membership and Voluntarily Join/Leave

In a legal execution, only the user is privileged to change his/hers membership status in a group. Such a change occurs in response to the application requests. Here we describe how, in a legal execution, processor p_i may join (leave) a group g by locally setting (resetting, respectively) $member_i$.

We use the self-stabilizing β-synchronizer algorithm [14] to coordinate view updates. The β-synchronizer is designed to be executed on a spanning tree of the system, in our case $\mathcal{T}$. There are two alternating phases for the β-synchronizer, propagation phase and convergecast phase. In a legal execution, processor p_l (the root of $\mathcal{T}$) is responsible for the membership updates. During the propagation phase, p_l propagates the view it maintains v_l. As v_l propagates through $\mathcal{T}$, every processor p_i assigns v_l to a local variable v_i that maintains its view. The value of $member_i$ of every processor p_i is accumulated during the convergecast phase. The value of $member_l$ of a leaf in $\mathcal{T}$ is delivered to p_k the parent of p_l. A parent of a leaf processor p_k, concatenates the values of the $member_c$, received from its children p_c, together with $member_k$ and delivers it to its parent, and so on. Once the convergecast phase terminates, the root sends the received concatenated information on the membership of all the processors, together with a view identifier (the view identifier is changed whenever the set of members is changed).

A transient fault detector monitors the consistency of the join/leave and membership information. Details are omitted from this extended abstract.

Before we turn to describe the actions taken upon a fault detection let us note that randomized transient failure detector can be used as well. In a legal execution, our deterministic transient failure detector repeatedly sends the same message through each link. Thus, the randomized technique proposed in [22], that uses a logarithmic size of the repeatedly sent message can be used here to further reduce the size of the messages sent. In such a case the failure detectors will detect a fault with high probability.

3.5 Fast Convergence

So far we have discussed transient fault detection, without describing the action taken when a fault is detected. The goal of the technique presented here is to ensure a fast convergence in the cost of a higher communication complexity. Once the transient fault detector detects a fault, we would like to activate the self-stabilizing tree update algorithm to regain consistency as soon as possible and then switch back to use transient fault detector.

Propagation of a Fault Detection: Once a fault is detected by a processor p_i, the processor propagates the fault indication to every other processor. Every tuple of the update tables is extended to include a *state* field, where the domain of the state is {*safe, dtc, act*}. We use the term the *source tuple* of p_i for the

single tuple of $\mathcal{TU}_i$ with i in the *id* field. p_i starts the propagation by assigning the values $\langle i, 0, nil, dtc \rangle$ in its source tuple. In the sequel we use the fourth field of p_i's source tuple as the state of p_i.

Every processor p_j that has at least one state field in a tuple of $\mathcal{TU}_j$ with a value not equal *safe* executes the update algorithm, sending messages through every attached link. When p_i sends the new value of $\mathcal{TU}_i$ to p_j and the state of p_j is *safe*, p_j changes it's state to *dtc*. The information on the fact that p_i detected a fault propagates to the entire system in the same way.

Our goal is to ensure that every processor p_k verifies that the tuples in the tables of the processors encode a fixed BFS tree rooted at p_k, and therefore the update algorithm is in a safe configuration. Then we allow the system to switch back to use the transient fault detector.

A central tool in achieving an indication on the completion of the reconstruction of the BFS trees is the *PIF* procedure. The propagation is done by flooding the system with the new information in the way we described above (for the case of *dtc*). The propagating processor, p_i, should receive a feedback on the completion of the propagation before finalizing the *PIF* procedure. The feedback is sent to a processor with a smaller distance from p_i, which p_i selects to be its parent in the tree. Every processor p_j uses the distance variable of the tuple with $id = i$ in $\mathcal{TU}_j$ as its (upper bound on the) distance to p_i.

A processor p_j sends a feedback only when the maximal distance difference of p_j to p_i, and the distance of any neighbor p_k of p_j to p_i is 1. The fact that the value in the distance fields is an upper bound on the distance from p_i guarantees that every neighbor p_j of p_i sends feedback when the value of its distance field is 1 and therefore has a fixed parent (namely, p_i). Moreover, p_j sends a feedback only when every of its neighbors has distance of at most 2. Thus, processors of distance 2 have the correct distance and therefore a fixed parent. Similar arguments hold for processors of greater distances, concluding that a fixed BFS rooted at p_i exists when p_i receives the feedback. More details can be found in [13]. We note that part of the new information that is propagated is a randomly chosen color that identifies (with high probability) the current *PIF* execution initiated by p_i, as a new *PIF* execution.

The fast convergence algorithm should ensure stabilization from an arbitrary state. We trace the activity of the system from the first fault detection. We would like the fault detection to ensure that every processor will start a *PIF* following the fault detection. Then, when every processor completes the *PIF* and verifies that its tree, is a fixed BFS tree we can stop executing the communication expensive tree update algorithm.

When p_i detected a fault it starts a *PIF* that causes every processor p_j either (1) to change a state from *safe* to *dtc* and start a *PIF* or (2) when p_j is in the state *act* to execute at least one more complete *PIF* before changing state to *safe*.

The update algorithm is executed by p_i whenever there exists a tuple in $\mathcal{TU}_i$ with a state field not equal *safe*. Otherwise, p_i responds to any $\mathcal{TU}_j$ message (sent by a neighbor p_j) by recomputing $\mathcal{TU}_i$ accordingly, and sending $\mathcal{TU}_i$ to p_j exclusively. (Note that the transient fault detector is disabled whenever there exists a tuple in $\mathcal{TU}_i$ with a state field not equal *safe*).

We may conclude that: once a transient fault is detected and propagated to the entire system it holds that (1) no processor is in a *safe* state, and (2) no transient fault is detected.

Upon Completing the Propagation of a Fault Detection: A processor p_i that has completed propagating the fault indication (completing a single *PIF*) changes state to *act*. Then p_i waits for all other processors to complete their propagation of fault detection, reaching a system state in which no processor (uses the failure detector to detect a fault and) starts propagating an indication of a failure. In other words, when p_i is in *act* state p_i repeatedly executes *PIF* until it receives an indication that no *dtc* tuple appears in any table.

The indication for the absence of *dtc* tuples, is collected using a *PIF* query. The *PIF* procedure is used to query the values of the state fields using the following procedure: Every tuple of the update tables is extended to include a *nodtc* bit field. When p_k chooses a new color, p_k set the *nodtc* bit $true$, and starts a *PIF*. A processor p_j sets the *nodtc* bit of every tuple in its table to $false$, whenever there exists a tuple with the state *dtc* in $\mathcal{TU}_j$. Whenever p_j sends feedback to its parent (as part of the *PIF*) p_j sends also the *and* result of the *nodtc* bit values of its children tables and its own table. Thus, a single $nodtc = false$ results in a $nodtc = false$ feedback that arrives to p_k.

We may conclude that once the *nodtc PIF* query procedure is completed with $nodtc = true$, then no processor is in a *dtc* state (and the transient fault detectors are disabled). Furthermore, let p_k be the first processor that changes its state from *act* to *safe*, after processor p_i had notified a fault detection. Let c be the configuration that immediately follows this state change of p_k. We will prove that, (1) the tree rooted at each processor in c is a fixed BFS tree, (2) the state field of every tuple in every table in c is *act* and, (3) no transient fault is detected.

Returning to Normal Operation: Once all the processors are in *act* state the system is ready to return to normal operation. A processor p_i changes state to a *safe* state when p_i is in *act* state and finds out that no *dtc* state exists in the system. Still p_i does not activate the transient failure detector until all processors change state to a *safe* state. p_i repeatedly executes *PIF* queries until it finds that the state of all the processors is *safe*. Thus, when a processor returns to use the transient failure detector all the processors are in a *safe* state and therefore a fault detection will result in a global state change to *dtc*, then to *act* and at last to *safe* after reaching a safe configuration.

The *PIF* query initiated by a processor in a *safe* state uses an additional *allsafe* bit field. When p_k chooses a new color, p_k set both the *nodtc* and the *allsafe* bits to $true$, and starts the *PIF* procedure. Recall that a processor p_j sets *nodtc* bit to $false$, whenever there exists a tuple with a *dtc* state in $\mathcal{TU}_j$. In addition, p_j sets the *allsafe* bit to $false$, whenever there exists a tuple with a state not equal to *safe* in $\mathcal{TU}_j$. p_k changes it state to *dtc* whenever there is a *dtc* tuple in $\mathcal{TU}_k$, or a feedback with $nodtc = false$ arrives. If the feedback carrying the *allsafe* bit is $true$, then p_k stops executing the update algorithm, and starts using the transient fault detector. If the *allsafe* bit is $false$ (and the *nodtc* bit is $true$) then p_k assigns $true$ to both *nodtc* and *allsafe* bits, and repeats executing the *PIF* query.

We note that the tree description used by the transient fault detector should be identical in all the processors before switching back to normal operation. Thus, the *allsafe* bit is also used to indicate that the tree description of a processor and its neighbors are identical (otherwise the *allsafe* bit that arrives in the feedback is $false$).

We may conclude that when the feedback of the *allsafe PIF* query is true, it holds that all the processors are in a *safe* state. Furthermore, let p_k be the first processor that returns to use the transient fault detector, after p_i propagated a fault detection. Let c be the configuration in which p_k returns to use the transient fault detector. Then in c it holds that the system is in a safe configuration with relation to the update algorithm.

We now turn to a detailed presentation of the fast convergence algorithm. The code of the fast convergence algorithm appears in Figure 4. In the code, we use the *PIF* and the *PIF query* procedures. A formal description of the *PIF* procedure can be found in [13]. The *PIF* procedure is extended to *PIF* queries (*nodtc* and *allsafe queries*) as described above.

Lines 1, 2 and 3 of the code describe the actions p_i takes according to it's state. When p_i is in a *dtc* state (line 1), p_i executes a *PIF* (line 1a), once the *PIF* is completed p_i changes its state to *act* (line 1b). When p_i is in *act* state (line 2), p_i repeatedly executes a *PIF query* to ensure that no *dtc* tuple exists in the system (line 2a). Then, p_i changes its state to *safe* (line 2b). In a *safe* state (line 3), p_i repeatedly executes a *PIF query* to ensure that all the states (of the processors and the state fields of the tuples) are *safe* states (line 3a). If there exists a *dtc* tuple, then p_i changes its state to *dtc* (line 3b). If indeed there are only *safe* tuples in the system then, p_i returns to use the transient failure detector (line 3c). Once the failure detector is operating, p_i changes its state to *dtc* when a fault is detected (line 3c and 3d).

```
1. state=dtc — (* Notify *)
   (a) Execute PIF
   (b) state ← act
2. state=act — (* Finish Notification *)
   (a) Execute PIF query
       until no dtc in the system
   (b) state ← safe
3. state=safe — (* Back to TFD *)
   (a) Execute PIF query
       until all safe or exists dtc
   (b) if PIF query results dtc then
       state ← dtc
   (c) else execute transient failure
       detector until fault detection
   (d) state ← dtc
```

Fig. 4. Fast convergence algorithm of p_i.

4 Concluding Remarks

This paper present the first asynchronous self-stabilizing group membership service. We believe that the new ideas presented in this paper will enrich the set of techniques used in the design of robust group communication services. For example, we do not utilize the idea of token passing for detecting a crash. Instead we present a self-stabilizing scheme that detects a fault fast (in a single asynchronous cycle) and is still communication efficient. Our membership service can serve as the base for additional group communication services.

Acknowledgment

We thank Idit Keidar, Nancy Lynch, Jennifer Welch and Alex Shvartsman for helpful discussions and suggestions, and Ilana Petraru for improving the presentation.

References

1. Y. Afek and G. M. Brown, "Self-stabilization over unreliable communication media," *Distributed Computing*, 7:27–34, 1993.
2. T. Anker, D. Breitgand, D. Dolev, and Z. Levy, "Congress: CONnection-oriented Group-address RES-olution Service" TR CS96-23, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, December 1996.
3. G. Alari and A. Ciuffoletti, "Group membership in a synchronous distributed system," *Proc. of the 5th IEEE Symposium on Parallel and Distributed Processing*, pp. 490-493, 1993.
4. Y. Afek, and S. Dolev, "Local Stabilizer," *Proc. of the 5th Israeli Symposium on Theory of Computing and Systems*, pp. 74-84, 1997.
5. A. Arora and S. Kulkarni, "Detectors and correctors: A theory of fault-tolerance components," *International Conference on Distributed Computing Systems*, pp. 436-443, 1998.
6. J. Beauquier, S. Delaet, S. Dolev, and S. Tixeuil, "Transient Fault Detectors" *Proc. of the 12th International Symposium on Distributed Computing*, Springer-Verlag LNCS:1499, pp. 62-74, 1998.
7. O. Babaoglu, R. Davoli, L. Giachini and M. Baker, "Relacs: A Communication Infrastructure for Constructing Reliable Applications in Large-Scale Distributed Systems," *Proc. Hawaii International Conference on Computer and System Science,* 1995, vol. II, pp. 612–621.
8. K.P. Birman and R. van Renesse, *Reliable Distributed Computing with the Isis Toolkit,* IEEE Computer Society Press, Los Alamitos, CA, 1994.
9. F. Cristian, "Reaching Agreement on Processor Group Membership in Synchronous Distributed Systems," *Distributed Computing*, vol. 4, no. 4, pp. 175-187, April 1991.
10. T. Chandra, V. Hadzilacos, S. Toueg, and B. Charron-Bost, "On the Impossibility of Group Membership," *Proc. ACM Symposium on Principles of Distributed Computing,* pp. 322-330, 1996.
11. E. W. Dijkstra, "Self-Stabilizing Systems in Spite of Distributed Control," *Communications of the ACM,* Vol. 17, No. 11, pp. 643-644, 1974.
12. S. Dolev, "Self-Stabilizing Routing and Related Protocols," *Journal of Parallel and Distributed Computing*, Vol. 42, pp. 122-127, May 1997.

13. S. Dolev, "Optimal Time Self-Stabilization in Uniform Dynamic Systems," *Parallel Processing Letters,* Vol. 8 No. 1, pp. 7-18, 1998
14. S. Dolev, *Self-Stabilization,* MIT Press, 2000.
15. S. Dolev and T. Herman, Superstabilizing protocols for dynamic distributed systems. *Chicago Journal of Theoretical Computer Science*, 1997.
16. R. De Prisco, A. Fekete, N. Lynch, and A.A. Shvartsman, "A Dynamic Primary Configuration Group Communication Service," *Proc. of the 13th International Conference on Distributed Computing (DISC)*, 1999.
17. D. Dolev and D. Malki, "The Transis Approach to High Availability Cluster Communications", *Comm. of the ACM*, vol. 39, no. 4, pp. 64-70, 1996.
18. Dolev, S., Schiller, E., "Communication Adaptive Self-Stabilizing Group Communication", Technical Report #00-02 Department of Computer Science Ben-Gurion University, Beer-Sheva, Israel, 2000.
19. P. Ezhilchelvan, R. Macedo and S. Shrivastava "Newtop: A fault-Tolerant Group Communication Protocol" in *Proc. of IEEE International Conference of Distributed Computing Systems*, pp. 296-306, 1995.
20. A. Fekete, N. Lynch and A. Shvartsman, "Specifying and Using a Partitionable Group Communication Service," *Proc. ACM Symposium on Principles of Distributed Computing,* pp. 53–62, 1997.
21. I. Keidar, J. Sussman, K. Marzullo, and D. Dolev "Moshe: A Group Membership Service for WANs". *MIT Technical Memorandum MIT-LCS-TM-593a*, revised September 2000.
22. E. Kushilevitz and N. Nisan *Communication Complexity*, Cambridge University Press 1998.
23. L.E. Moser, P.M. Melliar-Smith, D.A. Agarawal, R.K. Budhia and C.A. Lingley-Papadoplous, "Totem: A Fault-Tolerant Multicast Group Communication System", *Comm. of the ACM*, vol. 39, no. 4, pp. 54-63, 1996.
24. G. Neiger, "A new look at membership service", *Proc. of the 15th Annual ACM Symposium on Principles of Distributed Computing*, 1996.
25. R. van Renesse, K. P. Birman and S. Maffeis, "Horus:A Flexible Group Communication System," *Comm. of the ACM,* vol. 39, no. 4, pp. 76-83, 1996.
26. R. van Renesse, K. P. Birman, M. Hayden, A. Vasburd, and D. Karr, "Building adaptive systems using Ensemble," *Software-Practice and Experience,* 29(9):963-979, 1998.
27. R. van Renesse, Y. Minsky, and M. Hayden, "A gossip-style failure detection service", *In Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware '98)*, The Lake District, England, September 1998, pp. 55-70.
28. A. Segall, "Distributed Networks Protocols", *IEEE Trans. Comm.*, vol. IT-29, no. 1, pp. 23-35, Jan. 1983.