Distributed algorithms over communicating membrane systems

Gabriel Ciobanu

National University of Singapore, School of Computing
Department of Computer Science, gabriel@comp.nus.edu.sg

Abstract. This paper presents fundamental distributed algorithms over membrane systems with antiport carriers. We describe distributed algorithms for collecting and dispersing information, leader election in these systems, and mutual exclusion problem. Finally we consider membrane systems producing correct results despite some failures at some of the components or the communication links. We show that membrane systems with antiport carriers provide an appropriate model for distributed computing, particularly for message-passing algorithms interpreted here as membrane transport in both directions, namely when two chemicals behave as input and output messages and pass the membranes in both directions using antiport carriers.

Keywords: membrane systems, antiport interaction, distributed and parallel computing, broadcast, convergecast, flooding, leader election, mutual exclusion, fault tolerance, consensus.

1 Membrane computing

Membrane computing is based on *membrane systems* or P systems, a new class of distributed and parallel computing devices introduced by Păun (2000). The approach is based on hierarchical systems: finite cell-structures consisting of cell-membranes embedded in a main membrane called the *skin*. The membranes determine *regions* where *objects*, elements of a finite set, and evolution rules can be placed. The objects evolve according to given *rules* associated with a region. Objects may also move between regions. A *computation* starts from an initial configuration of the system, and terminates when no further rule can be applied. A software simulator of membrane systems is presented by Ciobanu and Paraschiv (2002).

A membrane structure is pictorially represented by a Venn diagram like the one in Figure 1, and it can be mathematically represented by a tree or by a string of matching parentheses. Hierarchical systems are well-known structures in computer science, and the notion of computation based on evolution rules is common. The interpretation of the computation is rather new: the result of a computation is a multiset of objects collected in the output cell or sent out of the system. The behaviour of the whole system is obtained by combining the

resulting multisets, or considering the multiplicity of objects present in a specific output membrane of a final configuration. Păun (2000) introduced three alternatives to look at membrane systems. Starting from these initial approaches, several variants were considered. Each of these variants has been shown to generate recursively enumerable sets. All these results emphasize a new pure computing paradigm inspired by a basic function of biomembranes.

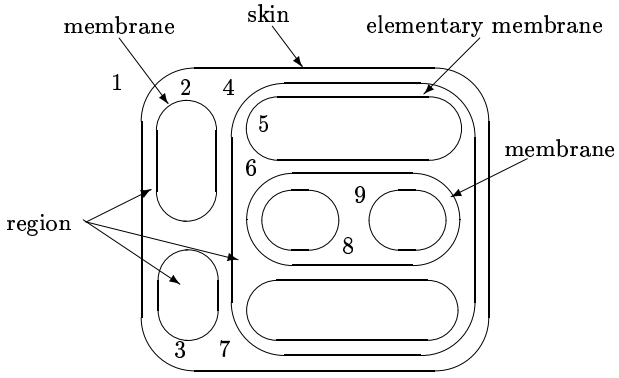


Fig. 1. *A* membrane structure

In this paper we refer to membrane system with antiport carriers; when two chemicals pass through a membrane in *both directions*, the process is mediated by an antiport carrier. Such a variant of membrane system was proposed by Păun and Păun (2002). The rules modeling this phenomenon are of the form $(x,\ out;\ y,\ in)$, where x, y are strings of symbols representing multisets of chemicals. The result of a successful computation is the number of objects present within the output membrane in the halting configuration. A computation which never halts yields no result. The set of all the numbers computed by Π is denoted by $N(\Pi)$. The family of all sets $N(\Pi)$, computed as above by systems Π of degree at most $m \geq 1$, is denoted by NPP_m. NRE denotes the family of recursively enumerable sets of natural numbers. It is proved that P systems with communication done by antiport rules are able to simulate Turing machines, namely $NPP_m = NRE$, for $m \geq 5$. The proof presented in Păun and Păun (2002) is based on matrix grammars with appearance checking. Related variants of membrane systems were then considered. This kind of results is common in the membrane computing community and it emphasizes the fact that the model is appropriate for computability results.

Ciobanu et al. (2003) present a new version of P systems called *Client–Server P Systems*. The new version is devoted to the interaction between components strongly dependent on their states and state transition. The client membranes are characterized by their states; the server membrane stores the clients states

and triggers the corresponding interaction rules. The communication rules are of symport type. A CSPS resembles the network client-server model by working with information and not using rules with creation/destruction of objects, but merely the power of communication. Ciobanu et al.(2003) show that CSPS have the same expressive power as Turing machines.

In order to derive a programming paradigm like the multiset processing, or a specification system, or a bio-programming language, it is necessary to have a larger framework and to emphasize some algorithmic features and software abilities of the membrane systems. This paper comes to cover a part of these requirements and reveals some distributed computing capabilities of the communicating membrane systems. An interesting aspect of the membrane systems is that they are distributed and parallel computing devices. To complement the overwhelming majority of researches in this area dealing with computability results, this paper aims to show that the membrane systems represent an appropriate model for distributed computing. We emphasize on the algorithmic aspects related to the distributed systems computational power provided by membrane systems. In membrane systems the process of passing objects through membranes in both directions is similar to message passing. We consider a system of communicating membranes with antiport carriers, and the main meaning regarding this choice is that the membranes *send* and *receive* information. We present some basic algorithms of distributed computing, starting with algorithms for broadcast, convergecast, flooding, leader election, mutual exclusion in distributed systems, the fault tolerant systems and the consensus problem.

2 Basic algorithms in membrane systems

The field of distributed computing is notoriously difficult, mainly due to uncertainties introduced by limited local knowledge, asynchrony, and failures. The fundamental issues underlying the design of distributed systems are related to communication, coordination, synchronization and fault tolerance. Mastering fundamental algorithmic ideas and techniques, someone is able to design correct distributed systems and applications. We present here some basic algorithms over communicating membrane systems, algorithms representing the core theory of distributed computing. For more information and notation about fundamental distributed algorithms, see Attiya and Welch (1998).

Collecting and dispersing information is central to any system; local information often has to be passed around in the system. This is where the message passing model becomes relevant. Two basic algorithms in any message passing model are *broadcast* and *convergecast*. Another common algorithm discussed in a message passing model is *flooding*. This algorithm constructs a spanning tree when a graph is given. The spanning tree provides in fact a conceptual link between distributed algorithms and membrane systems. Distributed algorithms collect and broadcast information in a network. Often they construct spanning trees of the network. The membrane systems themselves essentially have a (span-

ning) tree structure. Each membrane of the tree has exactly one parent, except the skin membrane which represents the root.

A membrane link represents a bidirectional channel, and information (given by objects) is required to be able to flow in both directions. Antiport carriers allow transmembrane transportation in both directions; this is the feature we consider as essential for antiport carriers. We introduce the antiport buffers. Every membrane link is modeled as an antiport channel with two buffers at each end: *antiport-in* and *antiport-out*. If membrane m_i and m_j are connected (which can only be the case if one is the parent of the other), then they share an antiport channel, and each membrane has two buffers; for instance, *antiport-in*$_{ij}$ and *antiport-out*$_{ij}$ represent the input and output buffers of the membrane m_i, and *antiport-in*$_{ji}$, *antiport-out*$_{ji}$ represent the input and output buffers of the membrane m_j. Note that *antiport-in*$_{ij}$ is equivalent to *antiport-out*$_{ji}$, since there is only one channel between any two membranes. Similarly, *antiport-out*$_{ij}$ is equivalent to *antiport-in*$_{ji}$. Thus, whatever is sent by m_i to m_j through *antiport-out*$_{ij}$ (the output buffer of m_i), it appears at *antiport-in*$_{ji}$ (the input buffer of m_j).

Broadcast: We consider a system in which a membrane m_i has to send some object to all the membranes of the system. Clearly, we have two cases of broadcast here - one in which m_i is the skin membrane (the root node), and secondly when m_i is any node. It is not difficult to see that the second case is a generalization of the first one. We start with the algorithm for the simple case (when m_i is the skin membrane), for easy understanding. Very often in a distributed computing algorithm, the root node has to broadcast a certain message. In a membrane system, the skin membrane has to distribute a certain object, say O, to all the membranes contained within. Here is a brief description of the algorithm: the skin membrane m_s sends first the object to all its children; upon receiving an object from its parent, a membrane sends the object to all its children, if any. Here is the formal pseudocode for this algorithm.

Algorithm: Skin Membrane Broadcast

`Initially` O `is in transit from` m_s `to all its children.`

`Code for` m_i $(0 \leq i \leq n-1)$:

1. `if` $i \neq s$, `receive` O `at` *antiport–in*$_{ij}$ `(where` m_j `is the parent membrane)`
2. `Send` O `to` *antiport–out*$_{ij_1}, \ldots,$ *antiport–out*$_{ij_k}$ `where` $m_{j_1}, \ldots, m_{j_k}$ `are the children membranes`
3. `terminate`

Analysis

Object Complexity. As evident from the above algorithm, the object is sent from a parent membrane to a child membrane exactly once. The algorithm terminates after sending O once to all its children. Thus, the total number of objects passed is equal to the number of edges in the spanning tree structure. Since, the number of edges in a spanning tree with n nodes is $n-1$, we obtain the result that $n-1$

objects are passed in a membrane system with n membranes. Object complexity of the algorithm is therefore $O(n)$.
Time Complexity. To understand the time complexity we first prove the following result:

Lemma 1. *In every admissible execution of the skin-broadcast algorithm, every membrane at level l (i.e. at a distance of l edges from the root node in the spanning tree) receives the object O in time l.*

Proof. The proof proceeds by induction on the distance l of a membrane from m_s. The basis is $l = 1$. From the description of the algorithm, each child membrane of m_s receives O from m_s in time 1.

We now assume that every membrane at level k ($k < l$, $l > 1$) receives the object O in time k. We need to show that every membrane m_i at distance l from the skin membrane m_s receives O in time l. Let m_j be the parent of m_i in the spanning tree, i.e. m_j contains membrane m_i. Since, m_j is at a distance $l-1$ from m_s, by the inductive hypothesis, m_j receives O in time $l-1$. By the description of the algorithm, m_j then sends the object to m_i in the next time step.

By the above lemma, a membrane system with l levels of membranes will have a time complexity of l. This corresponds to a depth of l in a spanning tree configuration. In the worst case, when the spanning tree is a chain, there can be at most $n-1$ levels for a system with n membranes. This shows that the time complexity of any membrane system for skin-broadcast is $O(n)$.
Therefore, *there is a skin-broadcast algorithm for membrane systems with object complexity $n-1$ and time complexity l, when l levels of membranes are present.*

2.1 Generalized broadcast

Each membrane m_i which needs to broadcast sends the object O to its parent and all children (if any). A membrane m_j, upon receiving an object from its parent, sends it to its children. If it receives the object from its child, then it sends the object to all its other children (if any), and parent. The pseudocode for this algorithm is given as follows.

Algorithm: Generalized Broadcast

```
Say membrane m_a, 0 ≤ a ≤ n − 1, needs to send the object to
all the membranes in the system.

Code for m_a:

1. if (a ≠ s)
2. Send O to its parent
3. Send O to antiport-out_{aj_1}, ..., antiport-out_{aj_k} where m_{j_1},...,m_{j_k}
   are the children membranes

Code for m_i, 0 ≤ i ≤ n − 1, i ≠ a:
```

4. Upon receiving O from its parent:
5. Send O to $antiport\text{-}out_{ij_1}, \ldots, antiport\text{-}out_{ij_k}$ where $m_{j_1}, \ldots, m_{j_k}$ are the children membranes
6. Upon receiving O from its child:
7. If$(i \neq s)$
8. Send O to $antiport\text{-}out_{ij}$ where m_j is its parent.
9. Send O to $antiport\text{-}out_{ij_1}, \ldots, antiport\text{-}out_{ij_{k-1}}$ where $m_{j_1}, \ldots, m_{j_k}$ are the children membranes and m_{j_k} is the sender

Analysis

Object Complexity. Similar to the skin-broadcast algorithm, the object is communicated on a parent-child membrane link exactly once. The communication on a given link is either initiated by a child or a parent, but not both. Thus, the total number of objects passed is equal to the number of edges in the spanning tree structure. Since, the number of edges in a spanning tree with n nodes is $n-1$, we obtain the result that $n-1$ objects are passed in a membrane system with n membranes. Object complexity of the algorithm is therefore $O(n)$. Thus, we see that object complexity of a generalized broadcast is equal to that of skin membrane broadcast.

Time Complexity. To understand the time complexity we shall first prove the following lemma:

Lemma 2. *In every admissible execution of the generalized broadcast algorithm, every membrane at level l (i.e. at a distance of l edges from the root node in the spanning tree) sends the object O to the root node in time l.*

Proof. The proof proceeds by induction on the distance l of a membrane from m_s. The basis is $l = 1$. From the description of the algorithm, each child membrane of m_s sends O to m_s in time 1.

We now assume that every membrane at level k $(k < l, l > 1)$ takes k time to send the object O to the skin membrane. We need to show that every membrane m_i at distance l from the skin membrane in the spanning tree sends O in time l. Let m_j be the parent of m_i in the spanning tree, i.e. m_j contains membrane m_i. Since, m_j is at a distance $l-1$ from m_s, by the inductive hypothesis, m_j sends O in $l-1$ time. By the description of the algorithm, m_j receives the object from m_i in 1 time step. Thus, l time units are needed to send the object from a membrane m_i to the skin membrane. Once the skin membrane receives an object, it takes l more time units by the lemma mentioned in skin-broadcast algorithm. Thus, a membrane system with l levels of membranes will have a worst case time complexity of $2 \times l$. In the worst case, when the spanning tree is a chain, there can be at most $n-1$ levels for a system with n membranes. This shows that the time complexity of any membrane system is $O(n)$.

Therefore, *there is a skin-broadcast algorithm for membrane systems with object complexity $n-1$ and time complexity $l+k$, when l levels of membranes are present and the membrane at k^{th} level broadcasts.*

2.2 Convergecast

The broadcast problem mentioned above aims at dispersing information held by a membrane to other membranes of the system. Convergecast, on the contrary, aims at collecting information from all the membranes of the system to the skin membrane. Many variants of the problem are available, for example, forwarding the sum of all the values held by membranes, or forwarding the maximum value, etc. In a general convergecast algorithm, instead of the result of a particular operation, all the values are forwarded. For simplicity we shall consider the algorithm of forwarding the sum of all the values held by membranes.

As it can be seen, unlike broadcast which is initiated by the membrane that wishes to disseminate information, convergecast is initiated by the *leaves*, i.e. elementary membranes, membranes which contain no inner membranes. This algorithm is recursive, and requires each membrane m_i to forward the sum of values held by its inner membranes. In other words, the sum of the sub-tree rooted at it. A membrane collects all the values held by its inner membranes, and computes the sum including its own values. This sum, s_i is then forwarded to its parent membrane. Clearly, each membrane has to receive a sum from each of its children before it can forward the sum to its parent. The pseudocode for the algorithm is given below.

Algorithm: Convergecast

`Code for leaf membranes:`

1. `Starts the algorithm by sending its value` x_i `to` $antiport\text{–}out_{ij}$ `where` m_j `is its parent.`

`Code for non-leaf membranes` m_i `with` k `children:`

2. `Receive` $s_{j_1}, \ldots, s_{j_k}$ `from` $antiport\text{–}in_{ij_1}, \ldots, antiport\text{–}in_{ij_k}$ `where` $m_{j_1}, \ldots, m_{j_k}$ `are the children membranes.`
3. `Computes` $s_i = x_i + s_{j_1} + \ldots + s_{j_k}$
4. `if` $(i \neq s)$`, sends` s_i `to` $antiport\text{–}out_{ij}$ `where` m_j `is its parent.`

Analysis

The analysis of this algorithm is analogous to the skin-broadcast algorithm, since the only difference in the two is the direction of object flow.

Object Complexity. As mentioned in the skin-broadcast algorithm, the object complexity of the algorithm is $n - 1$.

Time Complexity. The time complexity of the algorithm is $O(n)$, since at most $n - 1$ levels may be present in a membrane system with n membranes.

Therefore, *there is a convergecast algorithm for membrane systems with object complexity* $n - 1$ *and time complexity* l*, when* l *levels of membranes are present.*

3 Leader election in membrane systems

The existence of a leader in a membrane system can often simplify the task of coordination among the membranes. It might often be useful to have a leader

(other than the skin membrane as the default). It might also be the case that the criterion for leadership may not always be met by the skin membrane. The leadership election problem generally refers to the general class of symmetry breaking problems. The most general variant of it requires exactly one node from a system of many initially similar nodes to declare itself the *leader*, while the others recognize the leader and declare themselves *not-elected*. A membrane is anonymous if it does not have a unique identifier that can be used by a leader election algorithm. We have the following interesting result (for a biologist).

Theorem 1. *It is impossible to solve the leadership election problem in a system where the membranes are anonymous.*

The idea behind this impossibility result is that the symmetry between the membranes can be maintained forever if the membranes are anonymous (they are very similar). Without some initial asymmetry provided by unique identifiers, symmetry cannot be broken and it is impossible to elect a single leader: if one membrane is elected, then so are all the membranes. Therefore, we assume that every membrane in the system has one unique identifier *id*. An algorithm is said to be *uniform*, if it does not depend on the number of membranes. And conversely, *non-uniform* algorithms rely on the knowledge of the number of membranes.

3.1 A simple leader election algorithm

The most straightforward way to solve the problem is that every membrane sends an object with the maximum *id* among all its children (and itself) to its parent, and waits for a response from its parent. The skin membrane in turn, would reply to all its children with an object containing the maximum *id* that it received. Ultimately, one membrane (which receives its own *id* back) will be elected leader.

Algorithm: Simple Leader Election

For every membrane m_i

1. If membrane has no children, send id to $antiport\text{–}out_{ij}$ where m_j is the parent membrane
2. else
3. Receive $id_1, \ldots, id_k$ from $antiport\text{–}in_{ij_1}$, ..., $antiport\text{–}in_{ij_k}$, where m_{j_1}, ..., m_{j_k} are the children membranes.
4. Determine $leader = \max(id_1, \ldots, id_k)$
5. If $i = s$ (i.e. skin membrane)
6. Send $leader$ to $antiport\text{–}out_{ij_1}$, ..., $antiport\text{–}out_{ij_k}$ where m_{j_1}, ..., m_{j_k} are the children membranes
7. else
8. Send $leader$ to $antiport\text{–}out_{ij}$ where m_j is the parent membrane
9. Receive $leader$ from $antiport\text{–}in_{ij}$ where m_j is the parent membrane

10. Send *leader* to $antiport\text{-}out_{ij_1}$, ..., $antiport\text{-}out_{ij_k}$
    where m_{j_1}, ..., m_{j_k} are the children membranes.
11. Terminate as elected leader if $id = leader$.

Analysis
Object Complexity. The object complexity of the above algorithm is $O(n)$, since every link is used to exchange 2 objects. In a system with n membranes, there are $n-1$ such links. Thus, the object complexity is $O(n)$. The leadership algorithms in asynchronous rings have a lower bound of $O(nlogn)$. Whereas, in the synchronous case, an object complexity of $O(n)$ can be achieved at the cost of the time-complexity (Attiya and Welch (1998)). Generally, the membrane systems are structured like a tree, already providing a sufficient edge to break-symmetry as compared to a ring, where every substructure of the ring is symmetrical.

4 Mutual exclusion in shared memory

Shared memory is another major communication model and we shall see how membrane systems can be used effectively to model this as well. In a shared memory, processors communicate via a common memory area that contains a set of *shared variables*, which are also referred to as *registers*. In natural computing using membrane systems, membranes have access to some common resources and mutual exclusion can be modeled.

4.1 Formal model of shared memory systems

Before we proceed to understand mutual exclusion algorithms, we need to define the formal model for a shared memory system. We assume we have a system with n membranes, $m_0, m_1, \ldots, m_{n-1}$ and m shared variables or registers $R_0, \ldots, R_{m-1}$. Each register (shared variable) has a *type* which can specify the values which a register can take, the operations that can be performed on it, the value returned by each operation, and the updated value of the register as a result of the operation. Beside this, an initial value for the register has to be specified. Another important distinction in shared memory systems comes when analyzing algorithms. Unlike in object passing models, object complexity is meaningless. On the other hand, *space complexity* becomes relevant in this model. Space complexity can be measured in two ways: number of registers used, and number of distinct values the register can take. Measuring time complexity of shared memory algorithms is still a current research area and we only focus on whether the number of steps in the worst case running of the algorithm is infinite, finite, or bounded.

4.2 The mutual exclusion problem

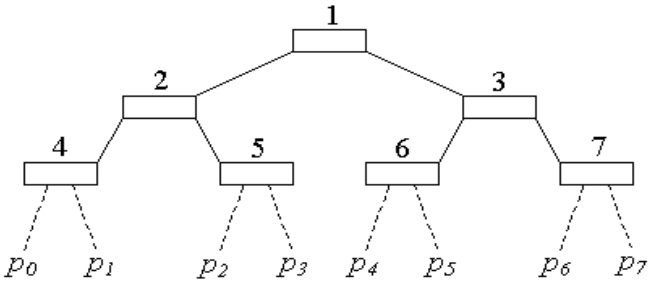
The *mutual exclusion* problem is one where different membranes need access to a shared resource that cannot be used simultaneously. *Critical Section* is a

code segment that has to be executed by at most one membrane at any time. *Deadlock* is a situation when one or more membranes are trying to gain access to a critical section, and none of them succeeds. *Lockout* is when a membrane trying to enter its critical section never succeeds. A membrane might need to execute some additional code segments before and after the critical section. This is to ensure mutual exclusion. We have four relevant sections of a mutual exclusion algorithm. *Entry* is a code section where the membrane prepares to enter critical section. *Critical Section* is a code section which has to be executed exclusively. *Exit* is the code section executed when a membrane leaves the critical section. *Remainder:* remainder of the code.

The desired properties are mutual exclusion, no deadlock and no lockout. *Mutual Exclusion* is achieved when in every configuration of every execution at most one membrane gets access to critical section. *No Deadlock* is achieved in every admissible execution, when membranes are in the *entry* section, at a later stage, a membrane is definitely in the critical section. *No Lockout* is achieved when in every admissible execution, a membrane trying to enter the critical section, eventually gets an entry.

4.3 Achieving mutual exclusion

The *test-and-set* and *read-modify-write* are powerful primitives used to achieve mutual exclusion. Another commonly known algorithm is the *Bakery algorithm* which uses *read/write* registers (Attiya and Welch (1998)). An appropriate algorithm for membrane systems would be the *tournament algorithm*. The conventional tournament algorithm can be modified to suit membrane systems. The tournament algorithm is a bounded mutual algorithm for n processors. It is based on selecting one among two processors at every stage, and thus selecting one among n processors in $\lceil \log_2 n \rceil$ stages. The algorithm is therefore recursive and every processor that succeeds at a stage climbs up the binary tree. The processor to reach the root gains entry to the critical section. An example with 8 processors is presented in the following picture.



Conceptually, membranes within a parent membrane can select one among themselves using the tournament algorithm, and the parent can then forward the request to its parent in turn. The one membrane that succeeds to reach the skin level gains entry to the critical section. The number of rounds k in this

algorithm will be equal to the number of levels l of the system. The pseudocode for the algorithm is mentioned below. The conventional algorithm is used by the term `tournament` and the list of membranes *list* is passed to it. The algorithm returns the *id* of the membrane that succeeds in the tournament algorithm.

The tournament algorithm

l represents the maximum depth of the system
m_j is the parent membrane of m_i.

```
1. for k = l down to 1
2.    1) all parent membranes m_i at depth k:
3.       if (list ≠ φ)
4.          w = tournament (list)
5.       else
6.          w = −1
7.       end if
8.       send w to antiport-out_ij where m_j is the parent membrane
9.    2) all leaves m_i at depth k:
10.      if (need access to critical section)
11.         w = i
12.      else
13.         w = −1
14.      end if
15.      send w to antiport-out_ij where m_j is the parent membrane
16.   3) all parent membranes m_i at depth k − 1
17.      for p = 1 to c (c is the number of children membranes)
18.         receive w_ip from antiport-in_ij_p where m_j_p is its child
19.         if (w_ip ≠ −1)
20.            add w_ip to list
21.      end for
22. end for
```

An important observation is that the first step will not be executed in the algorithm when $k = l$. This is because there are no parent membranes at depth l since that is the maximum depth of the tree. The above algorithm is ideal for membrane systems and provides mutual exclusion with no deadlock and no lockout.

5 Fault tolerant consensus

For a system to coordinate effectively, often it is essential that every membrane within the system agree on a common course of action. With the help of leadership election, and a subsequent broadcast/flooding, it is possible for a consensus to be reached. However, there are times when certain elements of a distributed system do not quite work the way they should. This section discusses the problem of reaching a consensus despite having failures within parts of the system,

i.e. the consensus problem and fault tolerance. This approach could be related to the possibility of controlling the membrane permeability.

In the distributed setting, many things might go wrong and lead to failure. The difference is that the failure of one component may not necessarily cause other processors to go down. This provides the hope that we can ensure the system working properly even in the presence of some failures. Much of the activity in this area is dedicated to the development of so-called *fault-tolerant* algorithms. Such algorithms are expected to go on executing properly and producing correct results despite of some failures at some of the components or the communication links.

5.1 The consensus problem

Consider a system in which each membrane m_i needs to coordinate with the rest of the membranes and choose a common course of action, i.e. agree upon a value for the variable *decision*. A solution to the consensus problem must guarantee

Termination: In every admissible execution, all the non-faulty membranes must eventually assign some value to *decision*.

Agreement: In every admissible execution, all the non-faulty membranes must not decide on conflicting values.

Validity: In every admissible execution, all non-faulty membranes must make the correct decision, i.e. must choose the correct value for *decision*. In other words, that if the consensus problem in question is choosing the maximum value from a certain set of numbers, then the *decision* must actually be the maximum value from the given set of input values.

Clearly the consensus problem is an important one, and the process would be disturbed in the presence of membranes which behave in an undesirable manner. However, within certain restrictions, it is possible to achieve a fault-tolerant consensus.

5.2 Failures

A failure is said to occur when a membrane behaves abnormally. There are two basic types of failures. *Simple failures* are when some membranes within the membrane system just stop functioning and do not ever recover, but wrong operations are never performed. The *Byzantine failures* are when some faulty membranes may behave in an unpredictable manner, contrary to a process which would help to reach a consensus.

The Simple Failure Case

The most important parameter which needs to be determined is f, the maximum number of membranes that can fail so that the consensus may still be achieved. Such a system is called an f-resilient system. According to Attiya and Welch (1998), we have the following results:

Lemma 3. *In every execution at the end of $f+1$ rounds, all non-faulty membranes have the same set of values to base their decision upon.*

Theorem 2. *It takes an upper bound of $f+1$ rounds to solve the consensus problem with simple failures in an f-resilient system.*

Algorithm: Consensus
Initially, every membrane m_j has some object x_j which it needs to send to all other processors and ultimately reach a consensus based on these values.

In every round k, $(1 \leq k \leq f+1)$, m_i behaves as follows:

1. Send x_i to $antiport\text{-}out_{ij}$, where m_j is the parent membrane
2. If m_i has children membranes, then wait to receive x_{j_k} from $antiport\text{-}in_{ij_1}$, ..., $antiport\text{-}in_{ij_n}$, where m_{j_1}, ..., m_{j_n} are children membranes
3. If m_i has children membranes, then send each received x_{j_k} to $antiport\text{-}out_{ij_1}$, ..., $antiport\text{-}out_{ij_n}$, where m_{j_1}, ..., m_{j_n} are children membranes
4. Wait to receive x_{j_1}, ..., x_{j_n} from m_k; where m_k is the parent membrane and m_{j_1}, ..., m_{j_n} are the children membranes.
5. Add x_{j_1}, ..., x_{j_n} to an array/vector V.
6. If $k = f+1$ make decision based on the values stored in V

The Byzantine Failure Case

This type of failure is more severe. The case is called the Byzantine failure because of a metaphorical description of a plan of action taken by several divisions of the Byzantine army (i.e. with traitors) to attack an enemy city (Lamport, Shostak and Pease (1982)). In the Byzantine case, the faulty membranes behave arbitrarily and even maliciously. In a f-resilient Byzantine system there exists a subset of at most f "Byzantine faulty" membranes. It becomes difficult to distinguish between a functional and a Byzantine faulty membrane, because unlike a membrane that crashes and simply stops sending objects, a Byzantine faulty membrane continues to send objects which may hamper the consensus process that requires membranes to agree on a common action based on their possible conflicting inputs. It is known that a consensus can be reached only if less than a third of the processors are Byzantine faulty processors (Lamport, Shostak and Pease (1982)). The result is true also for membranes.

Theorem 3. *In a system with n membranes, having f Byzantine membrane, there is no algorithm to solve the consensus problem if $n < 3f$.*

There is an algorithm for n membranes which solves the consensus problem in the presence of f Byzantine failures (Lamport, Shostak and Pease (1982), Attiya and Welch (1998)).

Theorem 4. *To reach consensus in an f-resilient system, every non-faulty membrane must send at least $f+1$ objects to all other non-faulty membranes to meet the requirements of the consensus problem.*

6 Example and implementation

In this section we present briefly an example describing an immune response system against virus attacks. This example is implemented using a membrane system library to emulate the main functions of a membrane system, and an MPI program that takes advantage of the highly parallel features provided by the model. Ciobanu, Desai and Kumar (2002) present this example and its implementation.

When a virus enters a cell, it tries to destroy the host cell and all the surrounding cells by periodical replication and propagation. The human immune system is in charge of producing suitable antibodies which can counter-attack the virus. In the event that the antibody is not present, it needs to be transported (through intercellular communication) from the cell it is produced, to the cell where it is required. The survival of the cell depends on the availability of the antibodies. The antibody must be present at the cell before the virus destroys it.

Given an initial membrane structure with the viruses and antibodies, it is useful to know that when an equilibrium is reached, whether all the cells are *clean* or some cells remain *infected*. From a pharmaceutical perspective, this tells us whether or not the membrane requires any external medicinal supply (or whether it is strong enough to resist the virus). Certain configurations worsen at every subsequent phases i.e. more viruses survive and antibodies slowly get depleted. The detection of such patterns in early stages would increase the chances of cleaning the cells.

We can represent the organism which the virus infects as a membrane system with the physical skin of the organism being represented as the skin membrane in the membrane system. The cellular-hierarchy of the organism is modeled as the tree-structure of the membranes. We focus on the viruses and the antibodies and their interaction in the system. These are represented by the objects in the systems, and their characteristics (type of antibody, the virus life) are the symbols of these objects. Applying a certain number of rules, and using the basic algorithms of leader election and synchronization, we are able to identify whether a cell is clean or infected (analogous to whether the virus wins or not), and to maintain the required balance of antibodies in order to attain a clean state by a right synchronized communication among membranes. Determining whether the virus will survive is not a trivial problem. Ciobanu, Desai and Kumar (2002) present conditions to determine desirable equilibrium states. The example is implemented using a membrane system library to simulate the main functions of a membrane system, and an MPI program (Snir et al. (1998)) that takes advantage of the highly parallel features provided by the model. The program uses leader election and synchronization algorithms.

7 Conclusion and related work

In order to enlarge the world of the membrane systems and to derive a programming paradigm as the multiset processing and gamma systems of Banâtre and Le

Métayer (1993), or a specification language, even a bioprogramming language, it is necessary to have a larger framework and to emphasize some algorithmic features and software abilities of the membrane systems.

Communicating membrane systems are interesting from several points of view: they have a precise biological inspiration, the computation is done by communication, and they are computationally complete. This paper comes to reveal some distributed computing capabilities of the membrane systems. We describe some distributed algorithms for collecting and dispersing information in these systems. Finally, inspired by the real life phenomena, we consider systems where some failures are allowed, providing various results related to the way of passing these failures and reaching a consensus despite having failures within parts of the system. Since P systems are close to biochemical mechanisms at a cellular level, this work may give some insights on how biosystems are able to "solve" a distributed problem. We present an example describing an immune response system against virus attacks. The example is implemented using a P system library to simulate the main functions of a P system, and an MPI program that takes advantage of the highly parallel features provided by the model. The program uses leader election and synchronization algorithms.

Acknowledgements

Many thanks to Akash Kumar and Rahul Desai (National University of Singapore) for their help, implementation and useful comments related to the topic of this paper. Thanks to the referees for their helpful remarks.

References

Alberts, B., Johnson, A., Lewis J., Raff, M., Roberts, K., Walter, P., 2002. *Molecular Biology of the Cell.* Garland Publisher, 4th edition.

Attiya, H., Welch, J., 1998. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*, McGraw-Hill.

Banâtre, J.-P., Le Métayer, D., 1993. Programming by multiset transformation, *Comm. of the ACM* vol.36, 98-111.

Ciobanu, G., Desai, R., Kumar, A., 2002. Membrane systems and distributed computing. In Păun, Gh., Zandron, C. (Eds.), *Pre-proceedings Workshop on Membrane Computing*, MolCoNet vol.1, pp. 145-162.

Ciobanu, G., Dumitriu, D., Huzum, D., Moruz, G., Tanasă, B., 2003. Client-Server P Systems in modeling molecular interaction. In Păun, Gh., Rozenberg, G., Salomaa, A., Zandron, C. (Eds.), *Membrane Computing 2002*, Lecture Notes in Computer Science, Springer, Berlin.

Ciobanu, G., Paraschiv, D., 2002. P System Software Simulator, *Fundamenta Informaticae* vol.49, 61-66.

Lamport, L., Shostak, R., Pease, M., 1982. The Byzantine generals problems. *ACM Trans. Program. Lang. Syst.* vol.4(3), 382-401.

Păun, Gh., 2000. Computing with membranes, *Journal of Computer and System Sciences*, vol.61, 108-143.

Păun, A., Păun, Gh., 2002. The power of communication: P systems with symport / antiport, *New Generation Computers*, to appear.

Snir, M., Otto, S., Huss-Lederman, S., Walker, D., Dongarra, J., 1998. *MPI–The Complete Reference vol.1, The MPI Core*, 2nd edition, MIT Press.