# A Formal Model for the Newscast Algorithm

Alberto Cuesta-Cañada

Ken Sharman
IEEE Member

Anna I. Esparcia-Alcázar
IEEE Member

***Abstract*—The newscast algorithm has the purpose of building a distributed computing system with a random topology where independent nodes can operate in a pure peer-to-peer fashion. The main advantages of this algorithm and the topology it builds are their robustness and efficiency to spread information. This article gives a formal input / output automata model for each node running this algorithm and thus for the entire network. The viability of the algorithm is proved and a solid basis for future development is laid.**[1]

***Index Terms*—Protocols, Communication Systems, Distributed Computing, Peer-to-peer Systems**

## I. Introduction

Peer-to-peer (P2P) networks [1] are a recent technique to create computational networks that don't depend on centralized servers. Every machine (peer) in a P2P network has the same hierarchical level as all the rest. Instead of depending on a centralized control that guides the behaviour of the network, the peers are engineered to show certain emergent properties. *Scalability* and *robustness* are some of the useful ones. Scalability means that the properties of something doesn't change with variations of size, thus a scalable network should show the same properties no matter if it is composed by a handful of nodes or billions of them. Robustness means that something doesn't change if parts of it stop working, a robust network should show the same properties even if a large percentage of its nodes disappear.

The behaviour and potential of this kind of networks is best understood when analyzed with the help of emergence theory [2]. An emergent property is one that comes from the interaction of many small entities, observable at the macroscopic levels. The immunologic system, the behaviour of ant colonies or evolution itself are clear examples of emergent systems. In words of John H. Holland, father of genetic algorithms, emergence is "getting much from little" [3]. It is possible to obtain large robust and efficient networks as emergent consequences from the interaction of simple peers.

The newscast algorithm was developed as a peer-to-peer protocol to disseminate information in large networks such as the internet. There is a good deal of information about it in a technical report from Jelasity and van Steen [4] and in many others, including [5], [6], [7].

[1]This work was supported by the Spanish Ministry of Education and Science under the project NADEWeb (Nuevos Algoritmos Distribuidos y Evolutivos en la Web), TIC2003-09481-C04.

The authors are with the Complex Adaptive Systems Group at the Instituto Tecnológico de Informática. Address: Instituto Tecnológico de Informática, Ciudad Politécnica de la Innovación, Building 8G, Universidad Politécnica de Valencia, Camino de Vera s/n, c.p. 46022, Valencia, España. Email: {alcueca,ken,anna}@iti.upv.es

The algorithm was effectively implemented in the DRM library, which was the communications layer of the DREAM project [8]. DRM has been used in many works since its apparition [9], [6] and more often as a part of the JEO evolutionary computation library [10], [11], [12].

Newscast relies on epidemics theory to spread information [13]. In the probabilistic models of real epidemics, each infected individual is expected to infect a number of different individuals following a probabilistic distribution. Thus the disease (or information) expands in a stochastic way, infecting the entire network very efficiently and without any centralized control. The infection vectors in newscast are structures called *contributions* than contain the address of its creator, the time of creation and an optional data structure to use in custom applications.

These contributions define a random topology between newscast nodes. These topologies are known by their robustness and the low average path length between any two nodes [14]. The edges in a newscast network are continuously shuffled to spread the information and mantain the topology when nodes join or leave.

The growth of computing networks makes it more and more difficult to manage them with centralized approaches. The largest of computing networks, the world wide web, has been able to grow because of its decentralized structure. Anyone wanting to build a simillarly sized network, he will need a decentralized algorithm to construct it. Each node in a newscast network is totally independent, thus this algorithm could be easily used for this end.

Even though there are many works on newscast, as theoretical research and software applications, there is a a lack of a solid basis on the algorithm. Such basis would be useful for example for its proposed development as the communications layer of a peer-to-peer computing platform [8] or to do research on the properties of the newscast graph and other similar distributed systems with emergent properties.

The objective of this work is to present a formal model of the newscast algorithm and prove some of its most basic properties for future reference.

In section II the newscast algorithm will be presented in three different formats: in pseudo-code, input/output automata model and data flow diagram. Some analysis on the immediate properties of the algorithm will be discussed in section III. Finally some conclusions will be detailed in section IV.

Fig. 1. Pseudo-Code for the Newscast Algorithm

**Data**:

$my$: Data from local node, is a structure that contains

$name$: The name of the node.

$contribution$: The contribution of the node.

$cache$: A cache of contributions.

$peer$: Data from a peer, has the same structure than $my$.

**Events**:

*timeout*: $\Delta t_i$ seconds have passed since last merging.

*receive_merging($n$)*: Some node named $n$ wants to merge its cache with the local node.

**Functions**:

*send_node_data($n$)*: Sends the name, contribution and cache of the local node to a node named $n$.

*receive_node_data($n$)*: Receives the name, contribution and cache from the node named $n$.

*choose_peer()*: Selects a random contribution from the local cache and returns the name of its owner.

*clean_cache($C$, $n_1$, $n_2$)*: Removes any contribution from nodes named $n_1$ or $n_2$ from the cache $C$.

*remove_old($C$)*: Searchs the cache $C$ for duplicated contributions and removes the older ones.

*cut_to_size($C$)*: Removes random contributions from the cache $C$ until it is smaller than $\hat{C}$.

*continue*: Skip the rest of the loop and continue to the next iteration.

+: Concatenates sets and/or elements.

**procedure** newscast():

**loop**

$e$ = event()

**if** $e$ is timeout **then**

$peer.name$ = choose_peer()

send_node_data($peer.name$)

$peer$ = receive_node_data($peer.name$)

$my.cache$ = merge($my$,$peer$)

**end if**

**if** $e$ is receive_merging($peer.name$) **then**

send_node_data($peer.name$)

$peer$ = receive_node_data($peer.name$)

$my.cache$ = merge($my$,$peer$)

**end if**

**end loop**

**procedure** merge(**in**: $my$, $peer$) **out**: $C_+$

$C_+$ = $my.cache$ + $peer.cache$

clean_cache($C_+$, $my.name$, $peer.name$)

remove_old($C_+$)

cut_to_size($C_+$)

$C_+$ = $C_+$ + $peer.contribution$

**return** $C_+$

## II. The Algorithm

A good resource to understand the algorithm is the technical report from Jelasity and van Steen [4]. Another one is the source code for the DRM library [15]. This section has been compiled from the study of these works.

### A. Informal Explanation

Let us start defining some concepts and structures.

A *node* is a running instance of the newscast algorithm. It is defined by a data structure that contains a *name*, a *contribution* and a *cache*.

The *name* of the node is unique and it is enough to locate it in the network.

A *contribution* is three-element structure that contains:

The unique *name* of the node that created it.

A *timestamp*, the instant at which the contribution was created.

An optional *news* item, which purpose is dependant on the application that we would want to build over the newscast network.

In the network created by this protocol, each contribution is equivalent to an *edge* from the node hosting it to the node which created the contribution.

A *cache* is a set which can contain up to a certain maximum number of contributions, called *maximum_cache_size* or $\hat{C}$. The cache from each node defines its neighbours and the information it has about them.

$\Delta t_i$ i the *refresh_rate* is a fixed time period that a node waits from its last outgoing newscast communication to the next. The actions inside a *refresh_rate* amount of time are an *iteration*.

To explain the algorithm we'll start at the middle of its execution. As it is supposed to work forever, being in the middle of execution is by far the most common state. We have a network of nodes, each of them is in a different computer or thread and it has an own internal clock. Each node has also a cache with contributions of $\hat{C}$ other nodes. Let us concentrate on a single node, which we will call $my$.

After waiting for $\Delta t$ seconds since its last operation $my$ chooses a random contribution from its cache and sends its data to its owner. The node (named $peer$) at the other side of the channel reciprocates sending also its own data. These actions are pseudo-coded in the newscast procedure in the Fig. 1, taken from [4].

The merge procedure begins with $my$ joining both $my.cache$ and $peer.cache$ into a single structure which we call $C_+$. Then, it checks $C_+$ and removes any contributions either from itself or $peer$. In the next step, $my$ checks for duplicated contributions. Two contributions are said to be duplicated when they both belong to the same node, although the news item or the timestamp can be different. The matter of fact is, when two duplicated contributions are found, the one with the oldest timestamp is deleted. There can be only one contribution from a single node in a given cache and it should be the newest one.

The only remaining step to build the new $my.cache$ is to remove random contributions from $C_+$ until it is smaller than the defined $\hat{C}$ and add $peer.contribution$ to it. Then $my$ substitutes $my.cache$ for $C_+$. These actions are pseudo-coded in the merge procedure in the Fig. 1, developed from the source code in the DRM library [15].

The following happens from the point of view of $peer$. It suddenly receives a communication from $my$, who sends it its data. Then $peer$ sends back immediately its own data. Next $peer$ finishes what it was doing if needed and afterwards merges $my$'s data and its own cache as if the merging was initiated by itself.

Also, let us have a look at the behaviour of the algorithm at the start of its execution. A newscast node usually starts its execution by contacting an node which is known to be online or that is supplied by a user, like gnutella [16] and other pure peer-to-peer systems [1]. After that first merging the node will probably have a full cache and it will continue as has been explained in this section.

### *B. Input / Output Automata Model*

The input/output automaton model, developed by Lynch and Tuttle [17], is a labelled transition system model for components in asynchronous concurrent systems. The actions of an I/O automaton are classified as input, output and internal actions, where input actions are required to be always enabled.

The I/O automata model is used to prove properties of distributed systems in reasonable environments. By stating that the environtment must accomplish with some reasonable guidelines, we can accept erratic behaviour from the modelled system when these guidelines are not met. The model allows easy modularization and abstraction of the algorithms to prove its properties, we can divide the system into layers, or modules, and prove some properties separately before continuing to greater and more complex systems.

For the newscast algorithm, we are going to assume initially that the reasonable behaviour of the system is that the network has no transmission errors. This way we are going to prove that the distributed system created by an indefinite number of newscast nodes doesn't ever interlock.

Follows the I/O automata model for the newscast algorithm. It has four sections, the signature are the action names and its type (input, output or internal), the states are variables which can hold several values and determine the point of execution of the algorithm, the start are the initial values of the states and finally the actions are the things the algorithm will do depending on the states' values.

**Signature**:

(in) start_timeout
(in) receive_merging
(int) accept_merging
(out) propose_merging
(int) unqueue_peer_data
(int) join_cache
(int) clean_cache
(int) remove_old
(int) cut_to_size
(int) finish_merging

**States**:

*peer*: object with name, cache and contribution to store data about the far node.
*my*: object with name, cache and contribution that always stores the data about the local node. Can be considered constant.
$C_+$: temporary object to store the future my.cache
*timeout_warning*: if true, an outgoing communication waits to be done.
*pretending_peers*: A set of peers which sent their data to the local node for merging.

**Start**:

*my*:
*my.name* = The unique name of the local node, which never changes.
*my.cache* = null
*my.contribution*:
*my.contribution.name* = my.name
*my.contribution.news* = null
*my.contribution.timestamp* = local time at moment of creation
*peer* = null
$C_+$ = null
*timeout_warning* = false.
*pretending_peers* = $\emptyset$.

**Actions**:

start_timeout:
input:
$timeout$
action:
$timeout\_waiting \leftarrow true$
propose_merging:
precondition:
$timeout\_waiting = true$
$peer = null$
action:
$timeout\_waiting \leftarrow false$
$peer.name \leftarrow random(cache).name$
$send\_queue(peer.name) \leftarrow$
$\leftarrow send\_queue(peer.name) \cdot my$
output:
$merging(peer.name)$
comment:
$random(set)$ returns a random entry of the given $set$.
receive_merging:
input:
$merging(name)$
action:
$pretending\_peers \leftarrow pretending\_peers \cdot name$

$send\_queue(name) \leftarrow send\_queue(name) \cdot my$

accept_merging:

precondition:

$timeout\_waiting = false$

$peer = null$

$pretending\_peers = name \cdot pretending\_peers'$

action:

$peer.name \leftarrow name$

$pretending\_peers \leftarrow pretending\_peers'$

unqueue_peer_data:

precondition:

$peer.name \neq null$

$peer.cache = null$

$peer.contribution = null$

$recv\_queue(peer.name) =$

$= peer' \cdot recv\_queue(peer.name)'$

action:

$recv\_queue(peer.name) \leftarrow$

$\leftarrow recv\_queue(peer.name)'$

$peer.cache \leftarrow peer'.cache$

$peer.contribution \leftarrow peer'.contribution$

join_cache:

precondition:

$peer.name \neq null$

$peer.cache \neq null$

$peer.contribution \neq null$

$C_+ = null$

action:

$C_+ \leftarrow my.cache \cdot$

$\cdot\ peer.cache\ \cdot\ \{peer.contribution\}$

clean_cache:

precondition:

$C_+ \neq null$

$\exists\ c \in C_+\ /\ c.name = my.name$

action:

$C_+ \leftarrow C_+ - c$

remove_old:

precondition:

$C_+ \neq null$

$\exists\ c_1, c_2 \in C_+\ /\ c_1.name = c_2.name \wedge$

$\wedge\ c_1.timestamp \geq c_2.timestamp$

action:

$C_+ \leftarrow C_+ - c_2$

cut_to_size:

precondition:

$C_+ \neq null$

$\nexists\ c \in C_+\ /\ c.name = my.name$

$\nexists\ c_1, c_2 \in C_+\ /\ c_1.name = c_2.name \wedge$

$\wedge\ c_1.timestamp \geq c_2.timestamp$

$card(cache) > maximum\_cache\_size$

action:

$C_+ \leftarrow C_+ - random(cache)$

finish_merging:

precondition:

$C_+ \neq null$

$peer.name \neq null$

$peer.cache \neq null$

$peer.contribution \neq null$

$card(cache) \leq max\_cache$

$my.contribution \notin C_+$

$\nexists\ c \in C_+\ /\ c.name = my.name$

$\nexists\ c_1, c_2 \in C_+\ /\ c_1 \neq c_2 \wedge c_1.name = c_2.name$

$\forall c \in C_+ \Rightarrow c \in my.cache \bigcup$

$\bigcup\ peer.cache \bigcup \{peer.contribution\}$

$\forall c_1, c_2\ /\ c_1 \in C_+ \wedge c_2 \in my.cache \wedge$

$\wedge\ c_1.name = c_2.name \Rightarrow$

$\Rightarrow c_1.timestamp \geq c_2.timestamp$

$\forall c_1, c_2\ /\ c_1 \in C_+ \wedge c_2 \in peer.cache \wedge$

$\wedge\ c_1.name = c_2.name \Rightarrow$

$\Rightarrow c_1.timestamp \geq c_2.timestamp$

action:

$my.cache \leftarrow C_+$

$C_+ \leftarrow null$

$peer \leftarrow null$

The diagram in Fig. 2 has been deduced from the formal model and could help to better understand the flow of the algorithm, although it is not a substitution of the formal explanation. In this particular diagram states are represented as arrows and actions as boxes. You need to have some states enabled in a particular way to activate some action, which will produce a different set of states and could trigger other actions. If the label of a state could be inferred then it has been removed to avoid cluttering.

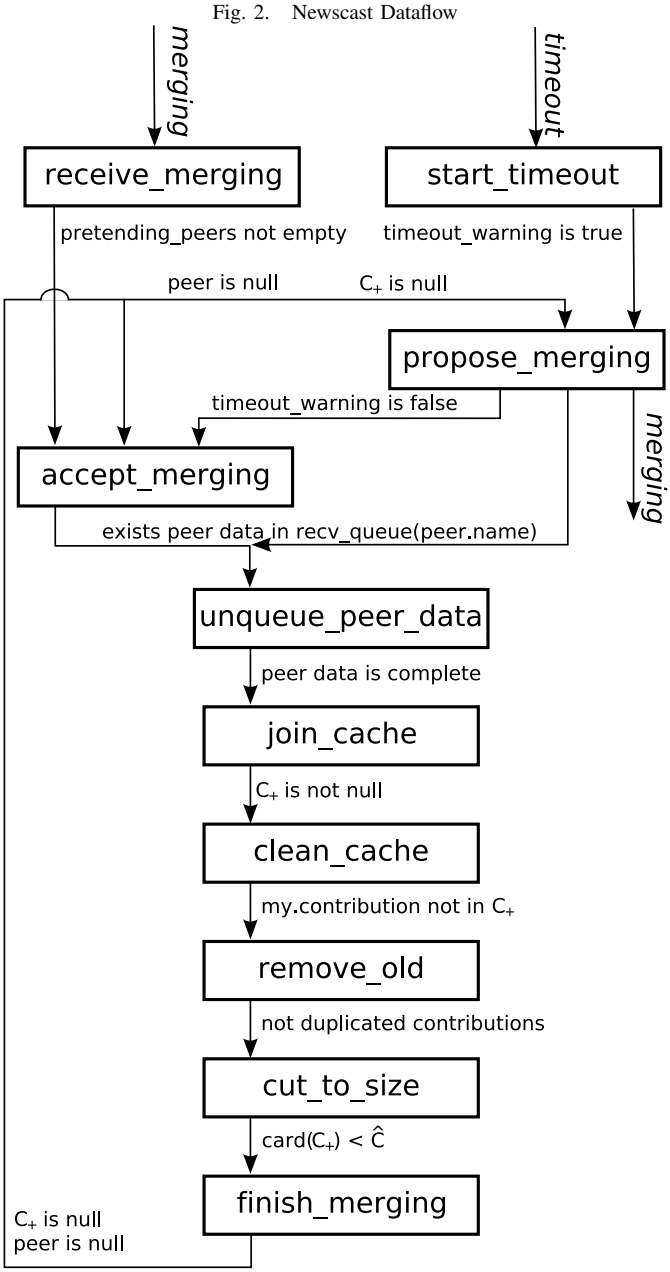
This diagram represents only partially the implications from the model and other behaviours not observable in the diagram could be inferred from the formal model. For example, even though in the diagram the clean_cache and the remove_old actions are sequential, they are not in the formal model. They have been drawn sequentially in the diagram because they seem easier to understand that way. Also these actions and cut_to_size should have looping arrows, as they are usually executed a small number of times until the finish_merging precondition is met.

In the Tuttle model, output actions in an automata are supposed to be input actions in another and share the same name, because they are the same action even if they happen in two different nodes and cause a different effect on each one of them. Instead of this we preferred to label differently the input and output parts of an action and connect them by a signal with a common name. An example can be seen in the diagram where the output action *propose_merging* is connected to the input action *receive_merging* by means of the *merging* signal.

The *timeout* signal which enables *start_timeout* is sent by the operative system. We found this easier to represent than enabling an inner clock in the automata.

Most actions have preconditions exclusive to them and

Fig. 2. Newscast Dataflow




they are changed during the action, thus not much confusion can arise of why the state arrows point from one particular action to another. There is one exception at the start of the timeout flow, where there is an arrow called *timeout_warning is true*. If the "timeout_warning" state is indeed true, then "timeout_warning is false" will be disabled. The opposite of course also applies.

Anyway, it should be clear that this data flow is here for illustrative purposes. The only reliable source when studying the newscast algorithm should be the formal model.

## III. Analysis of the Algorithm

The input / output automata model is used to formally prove properties of distributed algorithms and systems. These properties are usually divided into *liveness* properties, which state actions that will always be performed from certain inputs after a finite amount of time, and *safety* properties, which will be true in every state of the system.

### A. Safety: The Algorithm Never Gets Locked

The model doesn't allow an algorithm to block its inputs, so it seems logical that it will have always the opportunity to show some reaction to an input simultaneously to any other actions that it could be performing. In the newscast algorithm when a merging signal is received the node data (name, contribution and cache) is sent back by the same channel. To do this minimal computational logic is required, which should be present in any device with more than one possible communication channel. The cache is changed at the end of each cycle as an atomic action, so no corrupted information will be sent. Incoming peer data will be stored in the communication queue until there is time to get them processed. In the case of a timeout signal, the only reaction is to raise a informative flag.

We will now extract the first property of this algorithm. In a reasonable environment **the algorithm never gets locked**. Reasonable environment means that all the data elements are well formed, so actions after unqueue_peer_data will always end. Reasonable environment also means that sent data always arrives, that is, no network failures. To see that the algorithm doesn't lock is easy: There two actions competing for access to the cache data, propose_merging and accept_merging.

It is trivial to see that once the peer data has been unqueued in the unqueue_peer_data action the finish_merging state will be achieved. It is only a matter of deleting entries from a list which do not share certain properties.

Propose_merging depends on the states $timeout\_warning = true$ and $peer = null$. After the timeout_warning has been enabled, we know that if $peer \neq null$ it is because the algorithm is performing some action from accept_merging or propose_merging to finish_merging. As the environment guarantees that sent messages arrive, we know that the first possible precondition ($\exists\ cache \in recv\_queue(peer.name)$) will be met because the peer will always send its data. The environment also guarantees that the data is well-formed, so after that state the action finish_merging will be finally carried out, enabling propose_merging to continue. So there is no possible lock from being in the execution of start_timeout or propose_merging action.

From this explanation, we see that we must ask for another reasonable behaviour from the environment, the interval between timeout signals must be greater or equal than the time it takes to process peer data for a merging.

Accept_merging has three states as a precondition, $timeout\_warning = false$, $pretending\_peers \neq \emptyset$ and $peer = null$. The $timeout\_warning$ flag is enabled when

a timeout signal is received and disabled when the propose_action is performed. Given that $\Delta t_i$ is less than the time needed to process node data, this state will be eventually true at least from a propose_merging action to a finish_merging action. We have shown before that the finish_merging action will be carried out, so also the state of $peer = null$ will be true simultaneously to the previous stated precondition. Lastly, to see that pretending peers is not empty is trivial in a network of more than one node. Thus there is no possible lock from being in the execution of the receive_merging or accept_merging actions.

We can now extract another reasonable behaviour to request from the environment. If the time needed to execute a merging is $t_{ex.i}$, then the maximum rate at which new requests for merging can arrive is equal to $(\Delta t_i - t_{ex.i})^{-1}$. As we know from experimental settings [6], [4], this rate follows a Poisson distribution with mean equal to 1. Then we must ask the timeout to be $\Delta t_i \geq 2t_{ex.i}$.

So in an ideal environment with no network failures nor malicious actors, the algorithm will never remain in a single action, but every one will have some time to be executed. And if we adjust $\Delta t_i$ to reasonable values, the communication queues will remain short.

It is easy to extend this model to perform in a environment with network errors, enabling some mechanism to detect failed communications when performing the propose_merging or accept_merging actions which would discard that execution and return to a safe state. For example in the DRM java implementation [9] communications are performed through sockets, which incorporate a timeout mechanism which closes the socket and returns an exception when some time has passed without information in the channel. As each merging procedure is carried in a separate thread, the failed communication kills the thread with no consequences on the algorithm.

### B. Other Properties

It is interesting to analyze also the algorithm if we compress the entire algorithm to a single state and check its behaviour. Then we have a single-state automata with two inputs and an output. One of the inputs is the incoming merging signal, which gets accepted and shows no external reaction. The other input is from the internal clock, then the automata shows an outgoing merging in a time lapse no greater than $t_{ex.i}$ and no output will appear that is not preceded by a timeout signal.

This behaviour could also be modelled as safety and liveness properties (no output without a previous timeout, an output no more than $t_{ex.i}$ after a timeout, no reaction from an incoming merging). The proofs for this properties can be derived from the interlocking analysis. The resultant network structure is a set of nodes in a random topology, which send messages to random neighbours at given patterns, defined by their $\Delta t_i$.

It is useful to remember that the newscast algorithm is designed to host other applications and to provide them with communication tools. These other applications would have their own models and should prove their own liveness and safety properties.

## IV. Conclusions

In this report we have presented a formal description of the newscast algorithm. This formal description can be used to understand better the implementation of newscast in the DRM library and also to implement new applications based on newscast.

We showed that our model of the algorithm doesn't interlock given no malicious actors or failures in the communication network and that easy solutions are available to solve these problems. Also a minimal value was found for the frequency of the communications. This is important to build newscast networks which can be trusted as communication layers for general applications.

## References


[1] I. J. Taylor, *From P2P to Web Services and Grids: Peers in a Client/Server World*. London: Springer-Verlag, 2005.

[2] J. Goldstein, "Emergence as a construct: History and issues," *Emergence*, vol. 1, 1, pp. 49–72, 1999.

[3] J. H. Holland, *Emergence, from chaos to order*. UK: Oxford University Press, 2000.

[4] M. Jelasity and M. van Steen, "Large-scale newscast computing on the internet," Vrije Universiteit Amsterdam, Department of Computer Science, Tech. Rep., November 2002.

[5] S. Voulgaris, M. Jelasity, and M. van Steen, "A robust and scalable peer-to-peer gossiping protocol," in *Proceedings of the AP2PC 2003, LNAI*, Springer, Ed., vol. 2872, 2004, pp. 47–58.

[6] A. Cuesta-Cañada, "DRM newscast testing," Instituto Tecnológico de Informática, Valencia, Spain, Tech. Rep., October 2006.

[7] M. Jelasity, M. Preuß, and B. Paechter, "Maintaining connectivity in a scaleable and robust distributed environment," in *Proceedings of the IEEE International Symposium on Cluster Computing and the Grid*, I. Press, Ed., Berlin, May 2002, pp. 389–394.

[8] B. Paechter, T. Bäck, M. Schoenauer, M. Sebag, A. E. Eiben, J. J. Merelo, and T. C. Fogarty, "A distributed resource evolutionary algorithm machine (dream)," in *Proceedings of the Congress on Evolutionary Computation (CEC 2000)*, I. Press, Ed., 2000, pp. 951–958.

[9] M. Jelasity, M. Preuß, and B. Paechter, "A scalable and robust framework for distributed applications," in *Proceedings of the Congress on Evolutionary Computation (CEC 2002)*, I. Press, Ed., 2002, pp. 1540–1545.

[10] E. Alfaro-Cid, K. Sharman, and A. Esparcia-Alcázar, "A genetic programming approach for bankruptcy prediction using a highly unbalanced database," in *Proceedings of the First European Workshop on Evolutionary Computation in Finance and Economics (EvoFIN.07)*, ser. Lecture Notes in Computer Science. Valencia, Spain: Springer-Verlag, April 2007, accepted for publication.

[11] ——, "Evolving a learning machine by genetic programming," in *Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2006*, P. B. Gary Yen, Lipo Wang and S. Lucas, Eds., IEEE. Vancouver, Canada: Omnipress, July 2006, pp. 958–962, iSBN: 0-7803-9489-5.

[12] E. Alfaro-Cid, A. Esparcia-Alcázar, and K. Sharman, "Using distributed genetic programming to evolve classifiers for a brain computer interface," in *Proceedings of the 14 th European Symposium on Artificial Neural Networks, ESANN'06*. Bruges, Belgium: d-side publications, April 2006, pp. 59–64, iSBN 2 930307 06 4.

[13] P. T. Eugster, R. Guerraoui, A. M. Kermarrec, and L. Massoulie, "From epidemics to distributed computing," *IEEE Comput.*, 2003, available at http://agva.informatik.uni-kl.de/pgc/papers/epidemic-eugster.pdf.

[14] X. Wang and G. Chen, "Small-world, scale-free and beyond," *IEEE Circuits and Systems Magazine*, vol. 3, no. 2, pp. 6–20, 2003, available at http://www.ee.cityu.edu.hk/ gchen/pdf/Wang&Chen.pdf.

[15] "Dr-ea-m project," http://sourceforge.net/projects/dr-ea-m.

[16] "Gnutella project," http://www.gnutella.com.

[17] N. Lynch and M. Tuttle, "Hierarchical correctness proofs for distributed algorithms," in *Proceedings of the 6th Symposium on the Principles of Distributed Computing*, ACM, Ed., New York, 1987, pp. 137–151.