Constructing Resiliant Communication Infrastructure for Runtime Environments

George BOSILCA [a], Camille COTI [b], Thomas HERAULT [b] Pierre LEMARINIER [a] and Jack DONGARRA [a]

[a] *University of Tennessee Knoxville*
[b] *University of Tennessee Knoxville, Universite Paris Sud, INRIA*

Keywords. self-stabilization, binomial graph, scalability

1. Introduction

Next generation HPC platforms are expected to feature millions of cores distributed over hundreds of thousands of nodes, leading to scalability and fault-tolerance issues for both applications and runtime environments dedicated to run on such machines. Most parallel applications are developed using a communication API such as MPI, implemented in a library that runs on top of a dedicated runtime environment. Strong efforts have been made in the past decades to improve the performance, scalability and fault-tolerance at the library level. The most recent techniques propose to deal with failures locally, to avoid stopping and restarting the whole system. As a consequence, fault-tolerance becomes a critical property of the runtime environment.

A runtime environment is a service of a parallel system to start and monitor applications. It is deployed on the parallel system by a launching service, usually following a spanning tree to improve scalability of the deployment. The first task of the runtime environment is then to build its own communication infrastructure to synchronize the tasks of the parallel application.

A fault-tolerant runtime environment must detect failures, and coordinate with the application to recover from them. Communication infrastructures that are used today (e.g. trees and rings) are usually built in a centralized way and fail at providing support for fault-tolerance because a few failures lead with a high probability to disconnected components. Previous works [2] have demonstrated that the Binomial Graph topology (BMG) is a good candidate as a communication infrastructure for supporting both scalability and fault-tolerance for runtime environments. Roughly speaking, in a BMG, each process is the root of a binomial tree gathering all the processes.

In this paper, we present and analyze a self-stabilizing algorithm[1] to transform the underlying communication infrastructure provided by the launching service into a BMG, and maintain it in spite of failures. We demonstrate that this algorithm is scalable, tolerate transient failures, and adapt itself to topology changes.

[1] Self-stabilization systems [10] are systems that eventually exhibit a given global property, regardless of the system state at initialization

2. Related Work

The two main open source MPI library implementations, MPICH [4] and Open MPI [12] focus on performance, portability and scalability. For this latter purpose, both libraries manage on-demand connections between MPI processes, via their runtime environments. MPICH runtime environment, called MPD [8], connects runtime daemons processes through a ring topology. This topology is scalable in term of number of connection per daemon, but has two major drawbacks: two node failures is enough to disconnect nodes in two separate groups that cannot communicate anymore with one another, and communication information circulation does not scale well. The Open MPI runtime environment project, ORTE [9], deploys runtime daemons connected through various topologies, usually a tree. Recently, some works have proposed the integration of a binomial graph in ORTE [2]. However, the deployment of this topology inside ORTE is done via a specific node to centralize the contact information of all the other nodes and decide of the mapping of the BMG topology over ORTE daemons. This current implementation prevents scalability, and does not reconstruct the BMG upon failures. Our work focuses on the deployment and maintenance of a BMG topology in a distributed and fault-tolerant way, exhibiting more scalability.

Self-stabilization [14,10] is a well known technique for providing fault tolerance. The main idea of self-stabilization is the following: given a property $\mathcal{P}$ on the behavior of the system, the execution of a self-stabilizing algorithm eventually leads from *any* starting configuration, to a point in the execution in which $\mathcal{P}$ holds forever (assuming no outside event, like a failure). A direct and important consequence of this fault tolerance technique is that self-stabilizing algorithms are also self-tuning. No particular initialization is required to eventually obtain the targeted global property.

Some self-stabilizing algorithms already exist to build and maintain topologies. Most of them address ring [5] and spanning tree topologies [11], on top of a non-complete topology. They are usually designed in a shared memory model in which each node is assumed to know and be able to communicate with all its neighbors [1]. To the best of our knowledge, our work addresses for the first time building and maintaining a complex topology such as BMG. The classical shared memory model does not fit the actual systems we target in which connections are opened based on peer's information, thus we designed our algorithm using a message passing, knowledge-based, model [13].

3. Self-Adaptive BMG Overlay Network

We present in this section a self-stabilizing algorithm to build and maintain a binomial graph topology inside a runtime environment. This BMG construction supposes that every process in the system knows the connection information of a few other processes, at most one to be considered as its parent, such that the resulting complete topology is a tree of any shape. This assumption comes from the fact that the start-up of processes will usually follow a deployment tree. The connection information can be exposed to processes along their deployment, by giving to each process its parent's connection information according to the tree deployment. Each process then just contact its parent to complete the tree topology connectivity information.

The algorithm we propose is silent: in the absence of failure during an execution, the BMG topology does not change. This property is mandatory for being able to use this topology to route messages. We also focus on obtaining an optimal convergence time, in terms of number of synchronous steps, for underlying *binomial* trees, as the runtime

environment [7] we envisioned to implement this algorithm will usually deploy processes among such topology.

The construction of the BMG is done by the combination of two self-stabilizing algorithms. The first one builds an oriented ring from the underlying tree topology. The second one builds a BMG from the resulting ring. In the next subsections we present both algorithms, the key ideas of their proof of correctness and an evaluation of the time to obtain a BMG from different tree shape by simulation.

3.1. Model

System model Our algorithms are written for an asynchronous system in which each process has a unique identifier. In the rest of the paper, although process identifiers and actual processes are two different notions, we will refer to a process by its identifier. We assume the existence of a unidirectional link between each pair of processes. Each link has a capacity bounded by an unknown constant, and the set of links results in a complete connected graph. As in the knowledge network model, a process can send messages to another process if and only if it knows its identifier. When a process receives a message, it is provided with the sender's identifier. The process's identifiers can be seen as a mapping of IP addresses in a real-world system, and the complete graph as the virtual logical network connecting processes in such a system.

Algorithms are described using the guarded rules formalism. Each rule consists in a guard and a corresponding action. Guards are Boolean expressions on the state of the system or (exclusively) a reception of the first message available in an incoming link. If a guard is true, its action can be triggered by the scheduler. If the guard is a reception, the first message of the channel is consumed by the action. An action can modify the process's local state and/or send messages.

The state of a process is the collection of the values of its variables. The state of a link is the set of messages it contains. A configuration is defined as the state of the system, i.e. the collection of the states of every process and every link. A transition represents the activation of a guarded rule by the scheduler. An execution is defined as an alternate sequence of configurations and transitions, each configuration being obtained by the action of the triggered action in the transition on the previous configuration, in which the rule must hold.

We assume a centralized scheduler in the proof for the sake of simplicity. As no memory is shared between processes so that no two processes can directly interact, it is straightforward to use a distributed scheduler instead. We only consider fair schedulers, i.e. any transition whose guard remains true in an infinite number of consecutive configurations is eventually triggered.

Fault model We assume the same fault model as in the classical self-stabilization model: transient arbitrary failures. Thus, faults can result in node crash, message loss, message or memory corruption. The model of transient failures leads to consider that during an execution, there exists time intervals large enough so the execution converges to a correct state before the next sequence of failure. The consequence on the execution model is to consider no failure will happen after any initial configuration.

3.2. Algorithms

Remark 1 (Notations)**.** *We denote $\mathcal{ID}$ the identifiers of a process ; $List(c)$ a list of elements of type c, on which the operation $First(L)$ is defined to return the first element*

in the list L, and $next(e, L)$ is defined to return the element following e in the list L. Each of these functions return $\perp$ when the requested element cannot be found. $\perp$ is also used to denote a non-existing identifier.

Algorithm 1: Algorithm to build an oriented ring from any tree

Constants:
$Parent : \mathcal{ID}$
$Children : List(\mathcal{ID})$
$Id : \mathcal{ID}$
Variables:
$Pred : \mathcal{ID}$
$Succ : \mathcal{ID}$

1 - $Children \neq \emptyset \rightarrow$
$Succ = First(Children)$
Send $(F\_Connect, Id)$ **to** $Succ$

2 - **Recv** $(F\_Connect, I)$ **from** $p \rightarrow$
if $p = Parent$ **then** $Pred = I$

3 - $Children = \emptyset \rightarrow$
Send $(Info, Id)$ **to** $Parent$

4 - **Recv** $(Info, I)$ **from** $p \rightarrow$
if $p \in Children$ **then**
let $q = next(p, Children)$
if $q \neq \perp$ **then**
Send $(Ask\_Connect, I)$ **to** q
else
if $Parent \neq \perp$ **then**
Send $(Info, I)$ **to** $Parent$
else
$Pred = I$
Send $(B\_Connect, Id)$ **to** I

5 - **Recv** $(Ask\_Connect, I)$ **from** $p \rightarrow$
$Pred = I$
Send $(B\_Connect, Id)$ **to** I

6 - **Recv** $(B\_Connect, I)$ **from** $p \rightarrow$
$Succ = I$

Algorithm 2: Algorithm to build a BMG from a ring which size is known

Constants:
$Pred : \mathcal{ID}$
$Succ : \mathcal{ID}$
$N : integer$ size of the ring
$Id : \mathcal{ID}$
Variables:
`/* Clockwise links */`
$CW : Array[\mathcal{ID}]$
`/* Counterclockwise links */`
$CCW : Array[\mathcal{ID}]$

1 - $\perp \rightarrow$
$CW[0] = Succ$
$CCW[0] = Pred$
Send $(UP, CCW[0], 1)$ **to** $Succ$
Send $(DN, CW[0], 1)$ **to** $Pred$

2 - **Recv** $(UP, ident, nb\_hop)$ **from** $p \rightarrow$
$CCW[nb\_hop] = ident$
if $(2^{nb\_hop+1} < N)$ **then**
Send $(UP, ident,$ nb_hop + 1$)$ **to** $CW[nb\_hop]$
Send $(DN, CW[nb\_hop],$ nb_hop + 1$)$ **to** $ident$

3 - **Recv** $(DN, ident, nb\_hop)$ **from** $p \rightarrow$
$CW\_links[nb\_hop] = ident$
if $(2^{nb\_hop+1} \leq N)$ **then**
Send $(DN, ident,$ nb_hop + 1$)$ **to** $CCW[nb\_hop]$
Send $(UP, CCW[nb\_hop],$ nb_hop + 1$)$ **to** $ident$

3.3. Building a ring from a tree

The first step to build a binomial graph on top of a tree network consists in building a ring. This section defines a ring topology in our model and describes the proposed algorithm to build one from a tree. The last part of this section proposes a proof of correctness of this algorithm.

3.3.1. Topology description

Let $\mathcal{P}$ be the set of all the process identifiers of the system, $|\mathcal{P}| = N$ be the size of the system.

Tree topology For every process $p \in \mathcal{P}$, let $Parent_p$ be a process identifier in $\mathcal{P} \cup \{\perp\}$ that p knows as its parent. Let $Children_p$ be a list, possibly empty, of process identifiers from $\mathcal{P}$ that p knows as its children.

We define $anc_p(Q)$, the ancestry of the process p in the set of processes Q as a subset of Q such that $q \in anc_p(Q) \Leftrightarrow q \in Q \wedge (q = Parent_p \vee \exists q' \in anc_p(Q)$ s.t. $Parent_{q'} = q)$. A process p such that $Children_p = \emptyset$ is called a leaf. When $Children_p \neq \emptyset$, the first element of $Children_p$ is called first child of p, the last element of $Children_p$ is called the last child of p. We define the *rightmost leaf* of the set Q, noted 'rl_Q' as the unique leaf that is a last children process such that all processes in its ancestry in Q are last children processes.

A set of processes Q builds a tree rooted in r if and only if all processes of Q hold:

- $\forall p, q \in Q : parent_p = q \Leftrightarrow p \in Children_q$
- $Parent_r = \perp$
- $\forall p \neq r \in Q, r \in anc_p(Q)$

For the rest of the paper, we consider that for all configurations of all executions of the system, the collection of variables $Parent_p, Children_p$ for all processes builds a single tree holding all processes in the system. We call $root$ the process that is the root of this tree.

We define the subtree rooted in $r \in \mathcal{P}$, as the subset T_r of $\mathcal{P}$, such that $r \in T_r \wedge \forall p \in \mathcal{P}, r \in anc_p(\mathcal{P}) \Leftrightarrow p \in T_r$. Note that $T_{root} = \mathcal{P}$ is the largest subtree. The depth of a subtree T_r, noted $depth(T_r)$, is defined as the size of the largest ancestry in this subtree: $depth(T_r) = max\{|anc_p(T_r)|, p \in T_r\}$.

Ring topology For every process $p \in \mathcal{P}$, let $Pred_p$ and $Succ_p$ represent its knowledge of two process identifiers it considers as respectively its predecessor and its successor in the ring.

Definition 3.1. Consider the relation $\textcircled{s} : \mathcal{P} \times \mathcal{P}$ such that $p \textcircled{s} p'$ if and only if $Succ_p = p'$. We define $\mathcal{SU}_p$ as a subset of T_p such that $q \in \mathcal{SU}_p \Leftrightarrow q = p \vee p \textcircled{s} q \vee \exists q' \in \mathcal{SU}_p$ s.t. $q' \textcircled{s} q$.

Definition 3.2. Every process of the system is connected through a ring topology in a configuration C if and only if:

1. $\mathcal{P} = \mathcal{SU}_{root}$.
2. $\forall p \in \mathcal{P}, \exists q \in \mathcal{P}\ s.t.\ Succ_p = q \wedge Pred_q = p$
3. $Pred_{Succ_{root}} = Succ_{Pred_{root}} = root$

3.3.2. Algorithm description

We describe in this section the silent self-stabilizing algorithm 1 that builds an oriented ring from any kind of tree topology. Each process except the root of the tree knows a $Parent$ process identifier. Every process also has an ordered list of $Children$ process identifiers, possibly empty. The basic idea of this algorithm is to perform two independent and parallel tasks: the first one consists in coupling parents with their first child in order to build a set of chains of processes. The second one consists in coupling endpoints of every resulting chain.

The first task is performed by guarded rules 1 and 2. Rule 1 can be triggered by every process that have at least a child. When triggered, the process considers its first child as the next process in the ring by setting its $Succ$ variable to its first child identifier. It then sends a message to this first child to make it set up its $Pred$ variable accordingly. Rule 2 is triggered by reception of this information message and sets up the $Pred$ variable using

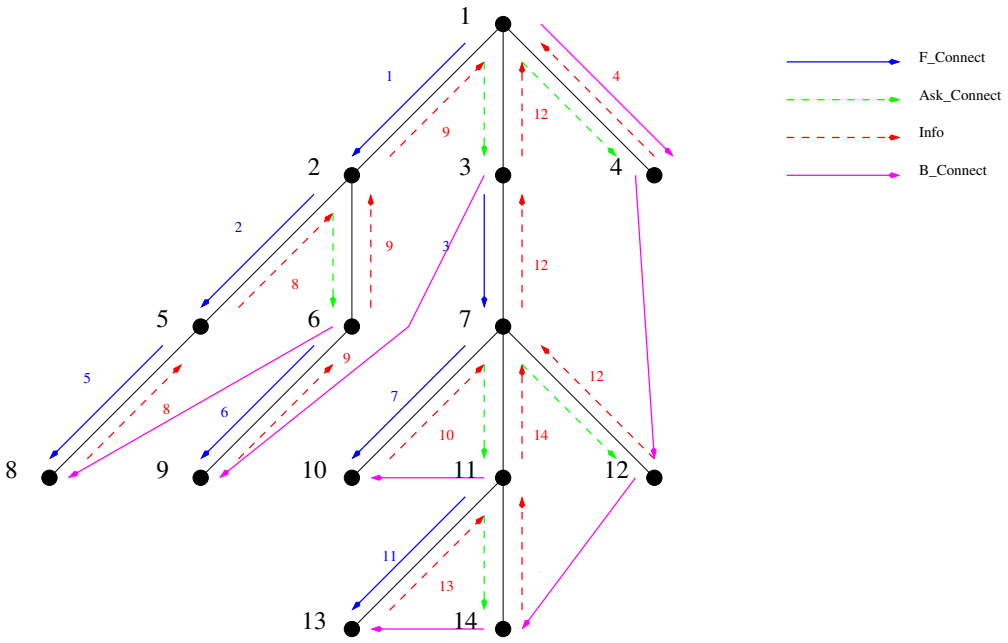


Figure 1. Message exchanged for building a ring on top of a tree

the identifier contained in the message. Note that each resulting chain eventually built by the first two rules has a tree leaf as one endpoint, and that every leaf of the tree is an endpoint of such a chain.

The second task consists in finding for each leaf a process among the tree, the first free sibling, to pick up as its successor in the ring. Rule 3 can only be triggered by leaf processes and sends a message $Info$ to their parent to find a process. Rule 4 describes what happens upon reception of such $Info$ message. When receiving $Info$ from a child c and c is not the last element of its $Children$ list, it looks for the process identifier c' that is the next element of c' in its $Children$ list. Then it sends an $Ask\_Connect$ message to c' containing the identifier c so that these two processes address each other (Rules 5 and 6). If c is the last element of the $Children$ list, then the process forwards $Info$ to its own parent if it has one, or acts as the process looked for if it is the root of the tree.

3.4. building a binomial graph from a ring

The next and final step to build a binomial graph on top of a tree overlay network consists in, starting from the ring topology constructed by algorithm 1, expanding the knowledge of every process with the process identifiers of their neighbors in the BMG to be obtained.

3.4.1. Topology description

As described in [3], a binomial graph is a particular circulant graph [6], i.e. a directed graph $G = (V, E)$, such that $|V| = |\mathcal{P}|$, $\forall p \in V$, $p \in \{0, 1, \ldots, |\mathcal{P}| - 1\}$. $\forall p \in V, \forall k \in \mathbb{N}\ s.t.\ 2^k < |\mathcal{P}|, \exists (p, (p \pm 2^k) mod\ |\mathcal{P}|) \in E$. It means that every node $p \in V$ has a clockwise (CW) array of links to nodes $CW_p = [(p+1)\ mod\ |\mathcal{P}|, (p+2)\ mod\ |\mathcal{P}|, \ldots, (p+2^k)\ mod\ |\mathcal{P}|]$ and a counterclockwise (CCW) array of links to nodes $CCW_p = [(p-1)\ mod\ |\mathcal{P}|, (p-2)\ mod\ |\mathcal{P}|, \ldots, (p-2^k)\ mod\ |\mathcal{P}|]$. It is impor-

tant to note that by definition, $\forall k > 0\ s.t.\ 2^k < |\mathcal{P}| : q = (p + 2^k)\ mod\ |\mathcal{P}| \in CW_p \Leftrightarrow q = (p + 2^{k-1} + 2^{k-1})\ mod\ |\mathcal{P}| \in CW_{p+2^{k-1}}$.

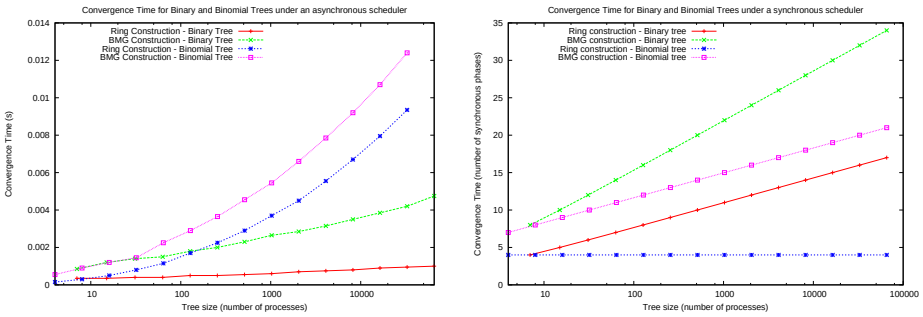
3.4.2. Algorithm description

The proposed algorithm uses the property of the BMG topology. Every node regularly introduces its direct neighbors to each other with rule 1. When a process is newly informed of its neighbor at distance 2^i along the ring, it stores this new identifier to the targeted list of neighbors, depending on the virtual direction, using either rule 2 or 3. Then it sends the identity of the processes at distance 2^i in both directions to introduce the two processes that are at distance 2^{i+1} along the ring to each other, unless $2^{i+1} \geq |\mathcal{P}|$.

4. Evaluation of the protocols

In this section, we present some simulations of the tree to ring and ring to BMG algorithms to evaluate the convergence time and communication costs of these protocols. The simulator is a ad-hoc, event-based simulator written in Java for the purpose of this evaluation. The simulator features two kind of scheduling: a) a synchronous scheduler, where in each simulation phase, each process executes fully its spontaneous rule if applicable, then consumes every messages in incoming channels, and executes the corresponding guarded rule (potentially deposing new messages to be consumed by the receivers in the next simulation phase), and b) an asynchronous scheduler, where for each simulation phase, each process either executes the spontaneous rule if applicable, or consumes one (and only one) message in one incoming channel, and executes the corresponding guarded rule (again, potentially deposing new messages to be consumed by receivers in another simulation phase). The asynchronous scheduler is meant to evaluate upper bound on convergence time, working under the assumption that although every process will work in parallel, the algorithms are communication-bound, and the total convergence time should be dominated by the longest dependency of message transmission. The simulator also features three kind of trees: 1) a binary tree, fully balanced and having its depth as a parameter; 2) a binomial tree, fully balanced and having its depth as a parameter; and 3) a random tree having both depth and maximal degree (each process of depth less than the requested depth having at least one child, and at most degree children) as parameter. For all simulations, every node starts with an underlying tree already defined (following the algorithms assumptions), and no other connection established ($Succ_p = Pred_p = CW_p[i] = CCW_p[i] = \perp, \forall p \in \mathcal{P}, \forall 0 \leq i \leq \log_2(N)$). Self-stabilizing algorithms cannot stop communicating, because a process could be initialized in a state where it believes that its role in the distributed system is completed. However, real implementations would rely on timers to circumvent this problem and use less resources when convergence is reached and no fault has been detected. To simulate this behavior, each process in our simulation becomes quiet (it deactivate its local spontaneous rule, but continues to react to message receptions) as soon as its local state is correct ($Succ$, $Pred$, $CW[0]$ and $CCW[0]$ are correctly set).

Figure 2(b) presents the convergence time of the tree to ring and ring to BMG algorithms under a synchronous scheduler, for the case of Binary and Binomial trees, as function of the size of the trees. The x-axis is represented on a logarithmic scale, and one can see that in the case of an underlying Binomial Tree, the convergence time of the tree-to-ring algorithm is 4 synchronous phases (each Info message originated at one leaf needs only to go up once to reach the parent of the tree this leaf is the rightmost leaf, thus the Info message can step down to the next children which exist and create a



(a) Asynchronous scheduler. (b) Synchronous scheduler.

Figure 2. Convergence Time for Binary and Binomial Trees.

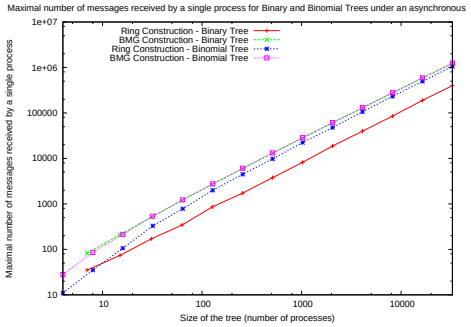
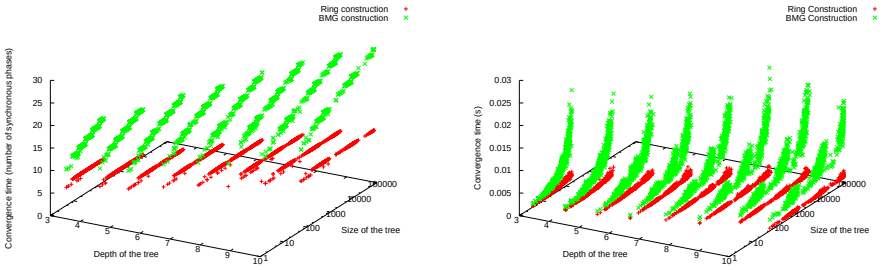


Figure 3. Maximal number of messages received by a single process for Binary and Binomial Trees under an asynchronous scheduler

$Ask\_Connect$ then a $B\_Connect$ message, hence 4 phases). For the case of a balanced Binary Tree, the longest $Info$ message has to go from one leaf in the "left" side of the tree up to the root, then two more messages to create the ring, hence $O(\log_2(N) + 2)$ phases. Until the ring has completely converge, exists at least one process in the system which can not start building one of its list of neighbors for the binomial graph. Thus it adds a $O(\log_2(N))$ more synchronous phases just after the ring is converged.

However, some nodes have to handle multiple communications during each phases, and communication-unbalance can happen. The consequence of this communication-unbalance is expressed in figure 2(a), that represents the same experiment under an asynchronous scheduler. With this scheduler, each simulation step consists of at most one message reception per process. Thus, if more messages have to be handled by some processes, the algorithms take significantly more time to reach convergence. To express convergence in time, we assume that each message takes 50 microseconds to be sent from one node to another (this time has been taken after measuring the communication latency of messages of 32 bytes between two computers through TCP over gigabit ethernet). As one can see on the figure, even if the projected convergence time remains very low for reasonably large trees (less than 1/50 of seconds for 64k nodes), the binomial tree presents a non-logarithmic convergence time, while in the case of binary tree, convergence time remains logarithmic. This is explained by figure 3, which presents the maxi-



(a) Synchronous scheduler.

(b) Asynchronous scheduler.

Figure 4. Convergence Time for Random Trees.

mal number of messages received by a single process during the convergence period for the Binary and the Binomial tree of same size. One can see that the number of messages received by a single process on a Binomial tree is much larger than for a Binary tree. Because a process removes one and only one message at a time from its message queue in the asynchronous scheduler, the size of the queue grows linearly with the number of direct neighbors and with time (as long as processes deposit new messages in the waiting queue). Thus, the waiting queue of the root in the binomial tree grows of $\log_2(N) - 1$ messages at each phase (until all leafs have ended generating $Info$ Messages), whereas it grows of 2 messages at each phase for a binary tree. Thus, convergence time of the binomial tree in this model is impacted by a factor $\log_2(N)$, and we can see in figure 2(a) that the convergence time for the binomial tree is indeed $\log_2^2(N)$, while it is $\log_2(N)$ for the binary tree.

The last two figures 4(a) and 4(b) present the convergence times (in number of phases, or in seconds for the asynchronous scheduler) as functions of the tree size and depth, for random trees. The synchronous version presents a logarithmic progression of the convergence time for the ring construction and for the binomial graph construction. The convergence time of the ring construction algorithm is not modified by the number of nodes in the tree, only by the depth of the tree itself. It presents an increase logarithmic in the depth of the tree, which is consistent with the theoretical analysis of the algorithm. Similarly, the BMG construction algorithm highly depends on the number of nodes in the tree: each process has to exchange $2\log_2(N)$ messages when the tree is built to build the finger table of the BMG, and this is represented in the figure. However, this progression remains logarithmic with the number of nodes. The asynchronous case is more complex to evaluate: because leafs become quiet only when their successor has received the $Ask\_Connect$ message causally dependent of their $Info$ Message, they introduce a lot of unnecessary $Info$ messages in the system. The asynchronous scheduler of the simulator takes one message after the other, following a FIFO ordering, and this introduces a significant slowdown of the $Info$ message, put in waiting queues. The projected time still remains very low, with less than 1/33 second for a 100k nodes tree. However, these results must be validated on a real implementation, to evaluate if the observed trend is due to simulation effects, or will be confirmed in a real-world system.

5. Conclusion

In this work, we present algorithms to build efficient communication infrastructures on top of existing communication trees for parallel runtime environments. The algorithms are scalable, in a sense that all process data storage, number of messages sent or received, and size of messages are logarithmic in the number of elements in the system, and the number of asynchronous rounds to build the communication infrastructure in the worst case is also logarithmic in the number of elements in the system. Moreover, the algorithms presented are fault-tolerant and self-adaptive using self-stabilization techniques. This paper presents a formal proof of the algorithms, and a performance evaluation based on simulations.

References

[1] Y Afek and A Bremler. Self-stabilizing unidirectional network algorithms by power supply. *Chicago Journal of Theoretical Computer Science*, 4(3):1–48, 1998.

[2] T. Angskun, G. Bosilca, and J. Dongarra. Binomial graph: A scalable and fault-tolerant logical network topology. In *Parallel and Distributed Processing and Applications, ISPA 2007*, volume 4742/2007 of *Lecture Notes in Computer Science*, pages 471–482. Springer Berlin / Heidelberg, 2007.

[3] T. Angskun, G. Bosilca, B. Vander Zanden, and J. Dongarra. Optimal routing in binomial graph networks. pages 363–370, December 2007.

[4] Argonne National Laboratory. MPICH2. `http://www.mcs.anl.gov/mpi/mpich2`.

[5] A Arora and A Singhai. Fault-tolerant reconfiguration of trees and rings in networks. *High Integrity Systems*, 1:375–384, 1995.

[6] J.-C. Bermond, F. Comellas, and D. F. Hsu. Distributed loop computer networks: a survey. *Journal of Parallel and Distributed Computing*, 24(1):2–10, January 1995.

[7] Darius Buntinas, George Bosilca, Richard L. Graham, Geoffroy Vallée, and Gregory R. Watson. A scalable tools communication infrastructure. In *Proceedings of the 6th Annual Symposium on OSCAR and HPC Cluster Systems*, June 2008.

[8] R. Butler, W. Gropp, and E. Lusk. A scalable process-management environment for parallel programs. In Jack Dongarra, Peter Kacsuk, and Norbert Podhorszki, editors, *Recent Advances in Parallel Virutal Machine and Message Passing Interface*, number 1908 in Springer Lecture Notes in Computer Science, pages 168–175, September 2000.

[9] R. H. Castain, T. S. Woodall, D. J. Daniel, J. M. Squyres, B. Barrett, and G .E. Fagg. The open run-time environment (OpenRTE): A transparent multi-cluster environment for high-performance computing. In *Proceedings, 12th European PVM/MPI Users' Group Meeting*, Sorrento, Italy, September 2005.

[10] S Dolev. *Self-Stabilization*. MIT Press, 2000.

[11] Felix C. Gärtner. A survey of self-stabilizing spanning-tree construction algorithms. Technical Report IC/2003/38, EPFL, Technical Reports in Computer and Communication Sciences, 2003.

[12] Richard L. Graham, Galen M. Shipman, Brian W. Barrett, Ralph H. Castain, George Bosilca, and Andrew Lumsdaine. Open MPI: A high-performance, heterogeneous MPI. In *Proceedings, Fifth International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks*, Barcelona, Spain, September 2006.

[13] Thomas Herault, Pierre Lemarinier, Olivier Peres, Laurence Pilard, and Joffroy Beauquier. A model for large scale self-stabilization. In IEEE International, editor, *Parallel and Distributed Processing Symposium. IPDPS 2007*, pages 1–10, march 2007.

[14] M Schneider. Self-stabilization. *ACM Computing Surveys*, 25(1):45–67, march 1993.

A. Appendix

A.0.3. Proof of correctness the ring construction (algorithm 1)

In this section we prove the correctness of the self-stabilizing algorithm 1 that connects every process in an oriented ring when they are connected in a tree based overlay. We first prove that every message in the starting configuration cannot be forwarded infinitely. We define a global property $\mathcal{P}$ on a configuration emphasizing the existence of a chain gathering all the processes and prove the correctness of this property. Then we prove the closure on configuration in which $\mathcal{P}$ holds. We prove the convergence from any starting configuration to a correct configuration by induction. We finally prove that the chain eventually connects to a ring.

Messages cannot be forwarded infinitely Let first define the direct causality $\Rightarrow$ between two messages m, m' in an execution as:

Definition A.1. $\forall m, m' : m \Rightarrow m'$ iff $\exists C, C'$ two configurations of the execution s.t. $C \rightarrow C',\ m \in C, m' \in C'$ and either $m = m'$ or $m \notin C' \wedge m' \notin C$

We define then the message causality $\prec$ in an execution as the transitive closure of direct causality. Intuitively $m \neq m' \wedge m \prec m'$ means that message m' is generated in the action of the guarded rule receiving m, or the consequence of a message generated in the action of this guarded rule. Now we can prove the following theorem staging that no message in the initial configuration, that could handle any kind of value and be in any link, will infinitely impact the execution:

Lemma A.1. *Let C_o be the initial configuration of an execution, $\forall m \in Messages(C_o), \exists C_i$ in the execution s.t. $\nexists m' \in Messages(C_i) : m \prec m'$.*

Proof. In the asynchronous model, every message in a link will eventually be received, and due to the fairness of the scheduler, the corresponding rule will eventually be triggered. The algorithm 1 exposes 4 types of message:

1. $F\_Connect$: triggers rule 2. No message is sent by this rule. When rule 2 is triggered by reception of a message f of type $F\_Connect$, which leads from configuration C to configuration C' such that $f \in C, f \notin C'\ \forall m \neq f \in C', m \in C$ since only f has been treated by rule 2, thus $\nexists m \in C'$ s.t. $f \Rightarrow m$.
2. $B\_Connect$: triggers rule 6, which does not result in the sending of any message either. Similarly, a reception of a message b of type $B\_Connect$, which leads from configuration C to configuration C' implies $\nexists m \in C'$ s.t. $b \Rightarrow m$.
3. $Ask\_Connect$: triggers rule 5, which results in sending one message of type $B\_Connect$ for which we have proved no message will causally follow.
4. $Info$: triggers rule 4, which results in sending one message of either $B\_Connect$, $Ask\_Connect$ or $Info$. For the first two types of message, we already prove that no message could causally follow their reception. When the reception of a message $Info$ results in sending a new message $Info$, it is addressed to the parent process in the tree, a constant throughout the execution. By induction on the parent link in a tree, this can be done only until the root process of the tree. Triggering rule 4 in the root process of the tree results on sending either a $B\_Connect$, $Ask\_Connect$, from which no messages can infinitely causally follow.

□

In order to prove that the algorithm eventually builds and maintains a ring, we proceed in two steps: 1) prove that the algorithm builds and maintains a chain gathering every process of the system bounded by the root process and the rightmost leaf of the tree. 2) prove that the two bounds of this chain eventually connect to a ring.

Definition A.2. A subtree T_r of processes forms a chain if the following property $\mathbb{P}$ on configurations holds: $\forall p \in T_r \setminus \{r\}, Succ_{Pred_p} = p \wedge \forall p \in T_r \setminus \{rl_{T_r}\}, Pred_{Succ_p} = p \wedge \mathcal{SU}_r = T_r$

Theorem A.1. *In every execution, starting from any configuration, exists a configuration starting from which the subtree T_{root} forms a chain that remain constant in the execution.*

Definition A.3 (Property $\mathbb{C}$)**.** A configuration holds the property $\mathbb{C}$ if and only if for every link $(p, q) \in \mathcal{P}^2$, the link contains only messages of type:

- l_1 if q is the first element of $Children_p$: $(F\_Connect, p)$
- l_2 if $q \in Children_p$ is successor of $q' \in Children_p$, and $q" = rl_{T_{q'}}$ being the rightmost leaf of the subtree rooted in q': $(Ask\_Connect, q")$
- l_3 if $Parent_p = q$, $q' = rl_{T_p}$ being the rightmost leaf of the subtree rooted in p : $(Info, q')$
- l_4 if $p = root$, q the rightmost leaf of T_{root}, $(B\_Connect, p)$
- l_5 if q is a leaf and $p \neq root$, let r be the root of the subtree of maximal depth s.t. q is the rightmost leaf of that subtree (i.e. $rl_{T_r} = q \wedge \forall r' \neq r \ s.t. \ rl_{T_{r'}} = q, \ depth(T_{r'}) < depth(T_r))$, $Parent_r = q'$, if $p = next(r, Children_{q'})$: $(B\_Connect, p)$
- l_6 else $(p, q) = \emptyset$

Definition A.4. A legitimate configuration is a configuration holding $\mathbb{C}$, in which for every process $p \in \mathcal{P}$:

- s_1 if $Children_p \neq \emptyset$, $Succ_p = first(Children_p)$
- s_2 if $Children_p = \emptyset$, let r_p be the root of the subtree of maximal depth s.t. p is the rightmost leaf of that subtree (i.e. $rl_{T_{r_p}} = p \wedge \forall r \neq r_p \ s.t. \ rl_{T_r} = p, \ depth(T_r) < depth(T_{r_p})$). Let $q = Parent_{r_p}$. If $q \neq \perp$, then $Succ_p = next(r_p, Children_q)$.
- s_3 if $p \neq root, \exists! q \ s.t. \ Succ_q = p \wedge Pred_p = q$

Lemma A.2 (Closure)**.** *Consider an execution $C_0 \rightarrow_0 C_1 \ldots$ If, for any i, C_i is a legitimate configuration, then C_{i+1} is a legitimate configuration.*

Proof. Let C_i be a legitimate configuration. We consider each guarded rule of the protocol and demonstrate that if the rule is applicable, the configuration obtained remains legitimate.

Rule 1: can be applied by any node which is not a leaf. According to s_1, in C_i, all non leaf processes have $Succ_p = First(Children_p)$. This rule keeps the successor to this value, and send a message $(F\_Connect, Succ_p)$, which is authorized by property l_1 of a legitimate configuration.

Rule 2: According to l_1, this rule can be triggered only by processes p such that $First(Children_q) = p$. It sets $Pred_p$ to q, which was already the case because of property s_3 held by C_i.

Rule 3: can be applied by leafs only. Let p be a leaf in C_i, it sends the message $(Info, p)$ to its parent, which is authorized by property l_3 (p is the rightmost leaf of the subtree holding p only).

Rule 4: can be applied by any process q that is not a leaf. The message $(Info, I)$ comes from a child p of q because of l_3. If p is the last children of q, then two cases arise: 1) if q is the root the tree, it sets $Pred_{root}$ to I, which can be anything for legitimate configurations, then sends $(B\_Connect, root)$ to I. In this case, due to rule l_3, $I = rl_{T_{root)}}$, and the message is authorized by rule l_4. In the other case 2), if q is not the root of the tree, it sends $(Info, I)$ to its parent. By definition of rightmost leaf, this means that $I = rl_{T_{Parent_q}}$, thus $(Info, I)$ is authorized in the channel $(q, Parent_q)$, because of rule l_3. Last case, if p is not the last children of q, then q sends $(Ask\_Connect, I)$ to $q' = next(p, Children_q)$. This message is authorized by rule l_2, because I is the rightmost leaf of the subtree rooted in p.

Rule 5: can be applied by process p that hold l_2: let q be the parent of p that sent the message $(Ask\_Connect, I)$, and q' be such that $p = next(q', Children_q)$. Because of rule l_2, $I = rl_{T_{q'}}$. The rule sets $Pred_p$ to $I = rl_{T_{q'}}$. In C_i, because of s_2, $Succ_I = p$, thus because of s_3, in C_i, $Pred_p = I$. Thus, the action of this rule has no effect on the state of p. Then p sends $(B\_Connect, p)$ to I, which is authorized by l_5.

Rule 6: can be applied by leaf p because of property l_5. p receives $(B\_Connect, I)$ where $I = next(r, Children_q), p = rl_{T_r}, q = Parent_r$, or $I = root$. If $I = root$, and $p = rl_{T_{root}}$. In this case, the value of $Succ_p$ does not determine a legitimate configuration. Otherwise, $Succ_p = I$ in C_i according to s_2. Thus, the configuration remains legitimate.

Moreover, no process sends a message in any other channel, thus l_6 holds. □

Corollary A.1. *Using the same reasoning, if C_i is a legitimate configuration, the application of any action in C_i leads to a legitimate configuration in which the state of the processes remain the same.*

Remark 2. *All channels hold the same message types during the whole execution starting from a legitimate configuration, and the state of the processes do not change during the execution, thus this algorithm is silent.*

Lemma A.3 (Correctness)**.** *Let $E = C_0 \longrightarrow_0 C_1 \ldots$ be an execution starting from a legitimate configuration C_0. $\mathbb{P}$ holds on C_0, for T_{root}, defining the chain of processes $p_1 \ldots p_n$, and the same chain $p_1 \ldots p_n$ remains connected in the rest of the execution.*

Proof. Property s_3 on the state of processes in legitimate configuration implies that $\forall p \in \mathcal{P}\backslash\{root\}, Succ_{Pred_p} = p$. Property s_3 also implies $\forall p \in \mathcal{P}\backslash\{root, rl_{T_{root}}\}, Pred_{Succ_p} = p$. Properties s_1 and s_3 implies $Pred_{Succ_{root}} = root$, thus $\forall p \in \mathcal{P}\backslash\{rl_{T_{root}}\}, Pred_{Succ_p} = p$. We prove that for any process p, $\mathcal{SU}_p = T_p$ by recursion on the subtrees of T_{root}. Let p be a leaf of T_{root}. $\mathcal{SU}_p = \{p\} = T_p$.

We assume that for any subtree T_p of depth at most $n \geq 0$, $\mathcal{SU}_p = T_p$. We prove that for any subtree $T_{p'}$ of depth $n + 1$, $\mathcal{SU}_{p'} = T_{p'}$. Consider the processes

$p_1, \ldots p_k$, children of p'. By assumption, $1 \leq i \leq k, \mathcal{SU}_{p_i} = T_{p_i}$. Because C_0 is a legitimate configuration, according to s_2, for $1 \leq i \leq k-1$, let $q_i = rl_{T_{p_i}}$, then $succ_{q_i} = p_{i+1}$. Moreover, because s_1, $succ_{p'} = p_1$. Thus, by definition of $\mathcal{SU}_{p'}$, $\mathcal{SU}_{p'} = \{p'\} \bigcup_{i=1}^{k} \mathcal{SU}_{p_i} = \{p'\} \bigcup_{i=1}^{k} T_{p_i} = T_{p'}$.

By induction on the subtrees of T_{root}, for any process p, $\mathcal{SU}_p = T_p$. Thus, $\mathcal{SU}_{root} = T_{root} = \mathcal{P}$. So, $\mathbb{P}$ holds on C_0. The corollary A.1 states that the chain of processes built in C_0 remains the same during the whole execution. □

Lemma A.4 (Convergence)**.** *Starting from any initial configuration, every execution eventually reaches a legitimate configuration.*

Proof. To prove the convergence, we first prove that starting from any initial configuration, every execution contains an unbound suffix into which every configuration verify $\mathbb{C}$. To prove this, we consider the execution $C_0 \rightarrow_0 \ldots C_k \ldots$, starting in any configuration C_0. Lemma A.1 implies that for some k, in C_k and onwards, no message of any channel causally depends on a message in C_0. Thus, all messages present in C_k and onwards is the result of the complete execution of a guarded rule, depending causally on the execution of a spontaneous rule (rules 1, 3).

Proving that the property l_1 holds in C_k and onwards is straightforward: this is a direct consequence of rule 1. It has already been proved that no $(F\_Connect, p)$ messages are not forwarded after their first reception.

The only rules that send $(Info, p)$ messages in channels are rule 3, and 4. These two rules send the message towards the parent. Moreover, rule 4 does not change the p parameter of the message; rule 3 puts the sender identifier in the message. When a process p sends $(Info, p)$ to its parent $p = rl_{T_p}$ (rule 3: p is a leaf). If (and only if) a process q forwards $(Info, p)$ to its parent (rule 4), it is because it receives it from its last child. Thus, $p = rl_{T_q}$. Thus, property l_3 holds in C_k and onwards.

Any $(Ask\_Connect, p)$ message causally depends on a $(Info, p)$ message according to rule 4 which is the only rule that sends such a message. Consider a process $q \neq root$ that sends a $(Ask\_Connect, p)$ message, causally depending on the reception of $(Info, p)$ from its child q', to q". If rule 4 sends a $(Ask\_Connect, p)$ message instead of a $(Info, p)$ message, it is because q' is not the last child of q, and q" is the next after q' in the children of q. Thus, $p = rl_{T'_q}$, and property l_2 holds in C_k and onwards.

Consider now that $root$ receives a $(Info, p)$ message from its last child q. For the same reason, $p = rl_{T_q}$, and rule 4 implies that $root$ sends $(B\_Connect, root)$ to p, and l_4 holds.

Last, consider what happens when a process $q \neq rl_{T_{root}}$ receives a $(B\_Connect, p)$ message. These messages causally depend on the reception of $(A\_Connect, q)$ message (rule 5). We already stated that any $(A\_Connect, q)$ message causally depends on a $(Info, q)$ message. Let q' be sender of the $(A\_Connect, q)$ message, $q' = Parent_p$, and there is a process q" such that p is the next child after q" in q'. Then, $q = rl_{T_{q"}}$, according to l_3. And as a consequence, l_5 holds in C_k and onwards.

All other channels are empty of any message, because no rule create another message than those that have been used. Thus, l_6 also holds, and $C_j, j \geq k$ verify $\mathbb{C}$.

In the suffix of the execution beginning in C_k, each message of the property $\mathbb{C}$ is generated an infinite number of time. Any message in the property $\mathbb{C}$ is an $(Info, p)$

message between a leaf and its parent, a $(F\_Connect, p)$ message between a node and its first child, or depends causally on such a message. The daemon being fair, $(Info, p)$ between a leaf and its parent, and $(F\_Connect, q)$ messages are generated an infinite number of time in the suffix (spontaneous rules 3 and 1). For the same reason, any message depending causally on a $(Info, p)$ message between a leaf and its parent is also eventually generated, and thus all messages of $\mathbb{C}$ are generated an infinite number of times in the suffix of the execution starting in C_k.

We can now prove lemma A.4: to prove this, we consider a configuration C_k of the execution $C_0 \rightarrow_0 \ldots C_k \rightarrow_k \ldots$ in which $\mathbb{C}$ holds and in which each message of the property $\mathbb{C}$ has been received at least once (such a configuration exists, as was proved above). We consider the messages that have been received before the configuration C_k:

All non-leaf processes p have sent a $(F\_Connect, p)$ message (l_1). Thus, according to rule 1, they have set $succ_p$ to their first child, and s_1 holds.

All leaf processes p have received $(B\_Connect, q)$ message(l_5). Thus, according to rule 6, they have set $succ_p$ to q, such that q is the next child after r in $parent_q$, where $rl_{T_r} = p$, and $depth(T_r) > depth(T_{r'})$ for any r' such that $p = rl_{T_{r'}}$. Thus, s_2 holds.

Moreover, no non-leaf process p can receive a $(B\_Connect, q)$ message. Indeed only rl_{T_r} processes (for some r), can receive such messages, thus non-leaf processes cannot. Moreover, they always send the same $(F\_Connect, p)$ message to their first child. Thus, $succ_p$ is always set to their first children in C_k. Leaf processes cannot send $(F\_Connect, p)$ messages, because they have no first child by definition, and they always receive the same $(B\_Connect, q)$ message because this $(B\_Connect, q)$ message causally depends only on the $(Info, p)$ message they have sent. Thus, $succ_p$ is always set to q, that is not a first child of a node. For any leaf p there exists a unique r such that $p = rl_{T_r} \wedge \forall r' \in \mathcal{P}$ s.t. $p = rl_{T_{r'}}, depth(T_{r'}) < depth(T_r)$. Two different leaves belong to two different subtrees of largest depth in which they the rightmost leaf. As a consequence, different leaves have different successors. Because this successor is not a first child of a node, their successors are different than the successors of non-leafs. Because the first child of any non leaf is different than the first child of any other non leaf, all successors of any process is different. $\forall p \in \mathcal{P} \setminus \{root\}, \exists! q, succ_q = p$.

$\forall p \neq root$, $Pred_p$ is assigned either when sending $(B\_Connect, q)$, or receiving $(F\_Connect, q)$. On the reception of $(F\_Connect, q)$, $Pred_p = q$ (rule 2). Any $(F\_Connect, q)$ message is sent by p after setting $Succ_q = p$. Thus, in C_k, for all process p that receives $(F\_Connect, q)$, $pred_{succ_p} = p$. When a process p sends $(B\_Connect, p)$ to a process q, it sets $Pred_p = q$ (rule 5). At the reception of this message, q sets $Succ_q = p$ (rule 6), thus $Pred_{Succ_p} = p$ for any $p \neq root$ sending $(B\_Connect, p)$. Thus $\forall p \in \mathcal{P} \setminus \{root\}, pred_{succ_p} = p$. So, . $\forall p \in \mathcal{P} \setminus \{root\}, \exists! q, succ_q = p \wedge pred_p = q$ (s_3).

Thus, C_kq is a legitimate configuration. □

We have proved that starting from any initial configuration we reach a configuration from which any configuration exhibits $\mathbb{P}$. We finally demonstrate that starting from any configuration, the root and the rightmost leaf of the tree eventually connect together.

Lemma A.5. *In any execution, $\exists C_i$ s.t $\forall C_j, i < j$, $Pred_{root} = rl_{T_{root}} \wedge Succ_{rl_{T_{root}}} = root$ in C_j*

Proof. From any initial configuration C_0, execution leads to a configuration C in which no message in C_0 impacts the system (theorem A.1), and in which a chain gathered all processes from the root to the rightmost leaf of the tree. Starting from C:

- if $Parent_{rl_{T_{root}}} = p_i$ then the link $(Parent_{rl_{T_{root}}}, p_i)$ only contains a finite number of messages $(Info, rl_{T_{root}})$ (property l_3), as the result of rule 3 eventually triggered.
- each process receiving a message $(Info, rl_{T_{root}})$, this process forward this message to its parent due to rule 4 and by definition of $rl_{T_{root}}$. Thus $root$ eventually receives $(Info, rl_{T_{root}})$.
- let $q = last(Children_{root})$. In configuration C and every following configuration of any possible execution, the link $(q, root)$ may contain only messages of type $(Info, rl_{T_q})$ (l_3). By definition $rl_{T_q} = rl_{T_{root}}$. When receiving such message by triggering rule 4, $root$ sets $Pred_{root} = rl_{T_{root}}$ and sends $(B\_Connect, root)$ to $rl_{T_{root}}$. Moreover it is the only way to set $Pred_{root}$, as rule 2 and 5 can only be triggered by receiving a message from the parent process.
- eventually $rl_{T_{root}}$ receives $(B\_Connect, root)$ and sets $Succ_{rl_{T_{root}}} = root$. Moreover, it is the only way to set $Succ_{rl_{T_{root}}}$ as rule 1 cannot be apply by leaf, and $\nexists q, r \in \mathcal{P}$ such that message $(B\_Connect, r) \in (q, rl_{T_{root}})$ due to property $\mathbb{P}$.

□

A.0.4. Proof of correctness of the BMG construction (algorithm 2)

For the sake of the proof, let name p_0 the root process of the initial tree from which the ring is built. let name p_i the process at distance i in the ring from p_o by following the $Succ$ links.

To prove that starting from any ring the algorithm eventually builds a BMG, we will first demonstrate that no message can be forwarded infinitely.

Lemma A.6. *For any execution,* $\forall i \in \mathbb{N}$, $\exists j > i \in \mathbb{N}\, s.t. \forall m \in Messages(C_i)$, $\nexists m' \in Message(C_j) : m \prec m'$

Proof. In the asynchronous model, every message in a link will eventually be received, and due to the fairness of the scheduler, the corresponding rule will eventually be triggered. The algorithm 2 exposes 2 types of message:

1. $(UP, ident, nb\_hop)$: triggers rule 2. By triggering this rule, a process sends two messages only if $2^{nb\_hop+1} < |mathcalP|$. Moreover both messages are sent with an incremented value for nb_hop. Thus such messages lead to apply rule 2 at most $log_2(|\mathcal{P}|)$ times.
2. $(DN, ident, nb\_hop)$: triggers rule 3 which is symmetrical to the rule 2. With the same reasoning as for UP messages, such messages lead to apply rule 3 at most $log_2(|\mathcal{P}|)$ times.

□

We now demonstrate that the algorithm eventually builds and maintains a BMG from a ring by induction.

Let define the property noted $Pr(i)$ on configuration, $i \in \mathbb{N}$:

Definition A.5. $Pr(i)$ holds in configuration C iff $\forall j \in \mathbb{N}, j \leq i$:

l_1: $\forall p_a \in \mathcal{P}, CW_{p_a}[j] = p_{(a+2^j)mod(|\mathcal{P}|)} \wedge CCW_{p_a}[j] = p_{(a-2^j)mod(|\mathcal{P}|)}$
l_2: $\forall p_a \in \mathcal{P}$, if $\exists (UP, p_a, j+1) \in Messages(C)$ then $(UP, p_a, j+1) \in (p_{(a+2^j)mod(log_2(|\mathcal{P}|))}, p_{(a+2^{j+1})mod(log_2(|\mathcal{P}|))})$
l_3: $\forall p_a \in \mathcal{P}$, if $\exists (DN, p_a, j+1) \in Messages(C)$ then $(DN, p_a, j+1) \in (p_{(a-2^j)mod(log_2(|\mathcal{P}|))}, p_{(a-2^{j+1})mod(log_2(|\mathcal{P}|))})$
l_4: $\forall p \notin \mathcal{P}, \nexists (UP, p, j+1) \in Messages(C) \wedge \nexists (DN, p, j+1) \in Messages(C)$

Definition A.6. Let $i \in \mathbb{N}$ such that $i < log_2(|\mathcal{P}|) \wedge i + 1 \geq log_2(|\mathcal{P}|)$. A legitimate configuration C is a configuration in which $Pr(i)$ holds.

Lemma A.7 (Closure)**.** *Consider an execution $C_0 \rightarrow_0 C_1 \ldots$ If, for any n, C_n is a legitimate configuration, then C_{n+1} is a legitimate configuration.*

Proof. Let first demonstrate that in any execution $C_0 \rightarrow_0 C_1 \ldots, \forall i < log_2(|\mathcal{P}|)$, if exists a configuration C_n in which $Pr(i)$ holds, then $\forall C_m, m > n, Pr(i)$ holds. We consider each guarded rule of the protocol and demonstrate that if the rule is applicable and triggered from a configuration in which $Pr(i)$ holds, then $Pr(i)$ holds in the resulting configuration.

Rule 1: can be triggered by any process $p_a \in \mathcal{P}$ in the system. This rule sets $CW_{p_a}[0]$ to $Succ_{p_a}$, which is by definition $p_{(a+2^0)mod(|\mathcal{P}|)}$ and $CCW_{p_a}[0]$ to $Pred_{p_a}$, which is by definition $p_{(a-2^0)mod(|\mathcal{P}|)}$. These setting are authorized by property l_1 of $Pr(i)$ and moreover are the same as in configuration C_i. Rule 1 adds one message $(UP, p_{(a-2^0)mod(|\mathcal{P}|)}, 1)$ to the link $(p_a, p_{(a+2^0)mod(|\mathcal{P}|)})$, which is authorized by property l_2, and adds one message $(DN, p_{(a+2^0)mod(|\mathcal{P}|)}, 1)$ to the link $(p_a, p_{(a-2^0)mod(|\mathcal{P}|)})$ which is authorized by property l_3. Note also that no other message are sent, which comply to property l_4. Thus, starting from any configuration in which $Pr(i)$ holds, the execution of rule 1 by any process leads to a configuration in which $Pr(i)$ holds.

Rule 2: can be triggered by any process p_b by receiving a message $(UP, p_{(a-2^j)mod(|\mathcal{P}|)}, j+1)$ from p_a, with $0 \leq j < i, j+1 < log_2(|\mathcal{P}|)$ and $b = (a+2^j)mod(|\mathcal{P}|)$, due to property l_2 that holds in configuration C_n. Note that if this rule is triggered by receiving any message $(UP, *, j+1)$ with $j \geq i$, then the process sets $CW_{p_b}[j+1]$ and $CCW_{p_b}[j+1]$ and sends message $(UP, , j+2)$ and $(DN, *, j+2)$, which are all not constraint by $Pr(i)$ and thus not affect the property. If $j < i$, this rule sets $CCW_{p_{(a+2^j)mod(|\mathcal{P}|)}}[j+1]$ to $p_{(a-2^j)mod(|\mathcal{P}|)}$. Let $k = j+1$, then

$$\begin{aligned} & b = (a+2^j)mod(|\mathcal{P}|) \\ \Leftrightarrow \quad & b = (a+2^{k-1})mod(|\mathcal{P}|) \\ \Leftrightarrow \quad & a = (b-2^{k-1})mod(|\mathcal{P}|) \end{aligned}$$

which leads to

$$\begin{aligned} & CCW_{p_{(a+2^j)mod(|\mathcal{P}|)}}[j+1] = p_{(a-2^j)mod(|\mathcal{P}|)} \\ \Leftrightarrow \quad & CCW_{p_{(a+2^{k-1})mod(|\mathcal{P}|)}}[k] = p_{(a-2^{k-1})mod(|\mathcal{P}|)} \\ \Leftrightarrow \quad & CCW_{p_b}[k] = p_{(b-2^{k-1}-2^{k-1})mod(|\mathcal{P}|)} \\ \Leftrightarrow \quad & CCW_{p_b}[k] = p_{(b-2^k)mod(|\mathcal{P}|)} \end{aligned}$$

which is authorized by property l_1, and are the same as in configuration C_n.
If $2^{j+2} < |\mathcal{P}|$, then p_b send one message $(UP, p_{(b-2^{j+1})mod(|\mathcal{P}|)}, j+2)$ in the link $(p_b, CW_{p_b}[j+1] = p_{(b+2^{j+1})mod(|\mathcal{P}|)})$, which is authorized by the prop-

erty l_2 as $j + 2 < log_2(|\mathcal{P}|)$, and send one message $(DN, CW_{p_b}[j+1] = p_{(b+2^{j+1})mod(|\mathcal{P}|)}, j+2)$ in the link $(p_b, p_{(b-2^{j+1})mod(|\mathcal{P}|)})$, which is authorized by the property l_3 as $j+2 < log_2(|\mathcal{P}|)$. Note that rule 2 does not imply any other send of message, thus comply with property l_4.

So, starting from any configuration in which $Pr(i)$ holds, the execution of rule 2 by any process leads to a configuration in which $Pr(i)$ holds.

Rule 3: is similar to rule 2, but for DN messages. The same reasoning is used to demonstrate that any process applying this rule from a configuration in which $Pr(i)$ holds leads to configuration in which $Pr(i)$ holds.

This is especially true for $i \in \mathbb{N}$ such that $i < log_2(|\mathcal{P}|) \wedge i + 1 \geq log_2(|\mathcal{P}|)$, thus starting from any legitimate configuration, the triggering of any rule by any process leads to a legitimate configuration. □

Lemma A.8 (Correctness)**.** *Let $E = C_0 \longrightarrow_0 C_1 \ldots$ be an execution starting from a legitimate configuration C_0. The definition A.6 holds on C_0, defining a BMG of processes from $\mathcal{P}$, and the same BMG remains connected in the rest of the execution.*

Proof. C_o is a legitimate configuration, where definition A.6 holds. Due to property l_1, $\forall j < log_2(|\mathcal{P}|) \in \mathbb{N}, \forall p_a \in \mathcal{P}, CW_{p_a}[j] = p_{(a+2^j)mod(|\mathcal{P}|)} \wedge CCW_{p_a}[j] = p_{(a-2^j)mod(|\mathcal{P}|)}$. which is the definition given in section 3.4.1 for a BMG. We have proved in lemma A.7 that for every $p \in mathcalP$, every $i < log_2(|\mathcal{P}|) \in \mathbb{N}, CW_p[i]$ and $CCW_p[i]$ remains the same, thus the every process remains connected through the same BMG for the whole execution. □

Lemma A.9 (Convergence)**.** *Starting from any initial configuration, every execution eventually reaches a legitimate configuration.*

Proof. Let first demonstrate that any execution eventually reaches a Configuration in which $Pr(0)$ holds. Consider the execution $C_0 \rightarrow_0 \ldots C_k \ldots$, starting in any configuration C_0. Due to lemma A.6, exists a configuration C_m in this execution such that $\forall m' \in Messages(C_0), \nexists m \in Message(C_m) : m' \prec m$. Thus in $C_m, \forall p_a \in \mathcal{P}$, any message $(UP, p_a, 1)$ or $(DN, p_a, 1)$ in $Messages(C_m)$ is only due to a process triggering rule 1 before in the execution, as this rule is the only one able to result in sending $(*, *, 1)$[2]. Starting from any configuration C_m, exists a configuration C_n in any execution $C_m \rightarrow_m C_{m+1} \rightarrow_{m+1} \cdots \rightarrow C_n \rightarrow_n$ such that each process $p_a \in \mathcal{P}$ has triggered at least once the rule 1 since C_m, due to the fairness of the scheduler. Any process $p_a \in \mathcal{P}$ when applying rule 1 sets $CW_{p_a}[0] = Succ_{p_a} = p_{(a+2^0)mod(|\mathcal{P}|)}$ and $CCW_{p_a}[0] = Pred_{p_a} = p_{(a-2^0)mod(|\mathcal{P}|)}$, which comply to l_1 and sends message $(UP, CCW_{p_a}[0], 1) = (UP, p_{(a-2^0)mod(|\mathcal{P}|)}, 1)$ in link $(p_a, p_{(a+2^0)mod(|\mathcal{P}|)})$ and $(DN, CW_{p_a}[0], 1) = (DN, p_{(a+2^0)mod(|\mathcal{P}|)}, 1)$ in link $(p_a, p_{(a-2^0)mod(|\mathcal{P}|)})$ which comply to l_2 and l_3. As these are the only messages sent by rule 1, and the process identity sent in those message are either $Pred_{p_a} \in \mathcal{P}$ or $Succ_{p_a} \in \mathcal{P}$, l_4 holds in C_n, thus $Pr(0)$ holds in C_n.

Now assume that in any execution $C_o \rightarrow_0 C_1 \rightarrow_1 \ldots$ exists a configuration C_n in which $Pr(i)$ holds for $i < log_2(|\mathcal{P}|) - 1$, and let demonstrate then that $\exists C_{r>n}$ a configuration in this execution in which $Pr(i+1)$ holds. We already demonstrated in

[2] $*$ represent any possible value

the proof of lemma A.7 that in every configuration $C_{s>n}$ $Pr(i)$ still holds. In any execution $C_n \rightarrow_n C_{n+1} \rightarrow_{n+1}$, every process $p_a \in \mathcal{P}$ eventually triggers rule 1 and sends message $(UP, *, 1)$ to $p_{(a+1)mod(|\mathcal{P}|)}$ and $(DN, *, 1)$ to $p_{(a-1)mod(|\mathcal{P}|)}$ due to fairness of scheduler. Thus every process eventually receives such message and triggers rule 2 and 3, which eventually leads to sends $(UP, *, 2)$ to $p_{(a+2)mod(|\mathcal{P}|)}$ and $(DN, *, 1)$ to $p_{(a-2)mod(|\mathcal{P}|)}$ due to fairness of scheduler, and so on and so forth. Thus eventually every process receives $(UP, *, i+1)$ and $(DN, *, i+1)$. Due to the induction hypothesis, for every process $p_a \in \mathcal{P}$, such messages are respectively $(UP, p_{(a-2^{i+1})mod(|mathcalP|)}, i+1)$ and $(DN, p_{(a+2^{i+1})mod(|mathcalP|)}, i+1)$ with respect to properties l_2 and l_3 of $Pr(i)$. Let C_s be a configuration reached by any execution $C_n \rightarrow_n C_{n+1} \rightarrow_{n+1}$ in which every process $p_a \in \mathcal{P}$ have triggered both rule 2 due to a message $(UP, p_{(a-2^{i+1})mod(|mathcalP|)}, i+1)$ and 3 due to a message $(DN, p_{(a+2^{i+1})mod(|mathcalP|)}, i+1)$, in any order at least once. Then in C_s, every process $p_a \in \mathcal{P}$ has $CW_{p_a}[i+1] = p_{(a+2^{i+1})mod(|mathcalP|)}$ due to rule 3 applied on the only possible message with $i+1$: $(DN, p_{(a+2^{i+1})mod(|mathcalP|)}, i+1)$, and $CCW_{p_a}[i+1] = p_{(a-2^{i+1})mod(|mathcalP|)}$ due to rule 2 applied on the only possible message with $i+1$: $(UP, p_{(a-2^{i+1})mod(|mathcalP|)}, i+1)$. As these messages have always the same value due to $Pr(i)$ holding in every configuration $C_{s>n}$, the value of $CCW_{p_a}[i+1]$ and $CCW_{p_a}[i+1]$ remains the same in every configuration $C_{t \geq s}$. Thus property l_1 of $Pr(i+1)$ holds.

Let $C_{u>s}$ be a configuration such that $\forall m' \in Messages(C_s), \nexists m \in Message(C_u) : m' \prec m$. Any execution reaches such configuration due to lemma A.6. Then in any configuration $C_{v \geq u}$, every message $(UP, *, i+2) \in Messages(C_v)$ comes from applying either rule 2 or rule 3.For every process $p_a \in \mathcal{P}$, both rules result in sending either $(UP, CCW_{p_a}[i+1], i+2) = (UP, p_{(a-2^{i+1})mod(|mathcalP|)}, i+2)$ in link $(p_a, p_{(a+2^{i+1})mod(|\mathcal{P}|)})$, or no message if $log_2(i+1) < log_2(|\mathcal{P}|)$, which comply with l_2 of $Pr(i+1)$, and either $(DN, CW_{p_a}[i+1], i+2) = (DN, p_{(a+2^{i+1})mod(|mathcalP|)}, i+2)$ in link $(p_a, p_{(a-2^{i+1})mod(|\mathcal{P}|)})$ or no message if $log_2(i+1) < log_2(|\mathcal{P}|)$, which comply with l_3 of $Pr(i+1)$. As no other message are sent by these rules, l_4 of $Pr(i+1)$ also holds in C_v, thus $Pr(i+1)$ holds in C_v.

Consequently, in any execution starting from any configuration, exists a configuration such that $i \in \mathbb{N}$, $i < log_2(|\mathcal{P}|) \wedge i+1 \geq log_2(|\mathcal{P}|)$, $Pr(i)$ holds. Thus such configuration is a legitimate configuration. □