

About decentralized swarms of asynchronous distributed cellular automata using inter-planetary file system's publish-subscribe experimental implementation

Vincent Manuceau

Department of Computer Science Research and Development, Makis Research, Remalard, Orne, France

Article Info

Article history:

Received Sep 6, 2021

Revised Dec 24, 2021

Accepted Jan 4, 2022

Keywords:

Cellular automata

Distributed cellular automata

Grid'5000

Inter-planetary file system

Publish-subscribe

ABSTRACT

This research describes the simple implementation of asynchronous distributed cellular automata and decentralized swarms of asynchronous distributed cellular automata built on top of inter-planetary file system's publish-subscribe (IPFS PubSub) experimentation. Various publish-subscribe (PubSub) models are described. As an illustration, two distributed versions and a decentralized swarm version of a 2D elementary cellular automaton are thoroughly detailed to highlight the simplicity of implementation with IPFS and the inner workings of these kinds of cellular automata (CA). Both algorithms were implemented, and experiments were conducted throughout five datacenters of Grid'5000 testbed in France to obtain preliminary performance results in terms of network bandwidth usage. This work is prior to implementing a large-scale decentralized epidemic propagation modeling and prediction system based upon asynchronous distributed cellular automata with application to the current pandemic of SARS-CoV-2 coronavirus disease 2019 (COVID-19).

This is an open access article under the [CC BY-SA](https://creativecommons.org/licenses/by-sa/4.0/) license.



Corresponding Author:

Vincent Manuceau

Department of Computer Science Research and Development, Makis Research c/o Creole Shop

7 Les Docks Saint-Marc, ZA Saint-Marc Sud, 61110 Remalard-en-Perche, France

Email: vincent@manuceau.net

1. INTRODUCTION

Decentralization and distribution of computing and communication systems represent an ever increasing topic of interest, with a wide array of problems to solve and an even wider array of applications from free-speech protection [1] to climate change modeling [2]. This article described a simple implementation of an asynchronous distributed cellular automata and a decentralized swarm of asynchronous distributed cellular automata built on top of inter-planetary file system's (IPFS) [3] publish-subscribe (PubSub) experimentation. After briefly describing notions of cellular automata (CA), IPFS, and PubSub protocols, two distributed versions and a decentralized swarm version of a simple 2D cellular automaton are detailed in order to highlight the simplicity of implementation with IPFS and the inner workings of these kinds of CA. The algorithms described in this paper are intentionally straightforward to focus on the simplicity of such asynchronous and decentralized PubSub systems. Implementation and preliminary results were obtained with experiments conducted across five datacenters throughout France with Grid'5000 testbed. This paper is motivated as preliminary work to build a very large scale distributed computing application that models population dynamics and predicts demography at city/region/country level, with multi-level and inter-level interactions, in the scope of the current SARS-CoV-2 coronavirus disease 2019 (COVID-19) pandemic.

2. DEFINITIONS

2.1. Cellular automata

A cellular automaton is a dynamical system composed of a finite lattice of cells with local and straight-forward communication capabilities. Each cell has a finite number of states evolving in discrete steps through time, and each cell state depends on the state of its neighbourhood. A transition function processes state changes. Originating from Ulam's works on dynamical systems [4], CA directly contributed to Von Neumann's theory of self-replicating automata [5]. CA and distributed cellular automata (DCA) represent an extensive field of study and notably interesting modelling and simulation tools for contemporary topics ranging from submicroscopic physics [6] to large scale macroscopic phenomena [7] such as climate change [2]. This paper will implement asynchronous distributed and decentralized swarms of cellular automata through IPFS PubSub capabilities.

2.2. Inter-planetary file system

Inter-planetary file system (IPFS) is a multipurpose, distributed, peer-to-peer, version-controlled file system with no single point of failure [3]. IPFS engenders a global Merkle directed acyclic graph data structure (Merkle DAG) [8], with a content-addressing storage block architecture driven by distributed hash tables (DHT), a block exchange system and a self-certifying namespace [3]. Stored content inside IPFS is accessible via content identifiers (CID) hyperlinks. Currently, IPFS is used for a wide variety of applications, such as distributed web applications and serverless applications [9], telecommunication, cloud data storage networks [10], content delivery networks (CDN), and blockchains [8], and even a crypto-currency based large scale decentralized file storage network called Filecoin [11], [12]. Its trustless and decentralized structure may lead to the development of a censorship-resistant and permanent web [1].

2.3. PubSub: IPFS and Libp2p implementations

2.3.1. About PubSub

Publish-subscribe models (PubSub) consists of asynchronous distributed and independent nodes, where each node can publish events or subscribe to related topics or contents through an overlay communication infrastructure [13]. Focusing on topic-based PubSub systems, when a node publishes an event to a related topic, all nodes that subscribed to this topic receive it asynchronously. Publishers and subscribers preserve their anonymity, as they do not interact directly and do not have to know each other to communicate. Specifically, the section below describes three topic-based PubSub implementations: IPFS FloodSub, Libp2p GossipSub, and Libp2p EpiSub.

2.3.2. IPFS FloodSub

FloodSub, also known as DumbSub or PubSub-Flood, is the first and the most simple PubSub implementation experiment in IPFS/Libp2p. FloodSub is based upon message routing by network flooding with no CastTree forming [14], and ambient peer discovery by the use of external distributed hash tables (DHT) [15]. Concerning message broadcasting on small networks, FloodSub has low latency and is ideal for instant messaging applications. However, it cannot scale to more significant sized networks due to its high overhead and bandwidth consumption.

2.3.3. Libp2p GossipSub

GossipSub is a gossip-based publication-subscribe protocol [16] relying on a mesh construction and a score function [17]. The network structure uses two types of bidirectional peering: full-message peerings, where peers send entire messages to sparsely connected peers called mesh members. The second type is metadata-only peerings, where peers gossip about message availability and maintain full-message peerings [18]. In a GossipSub network, any peer can change their peering type from full-message to metadata-only (pruning) and conversely (grafting). Additionally, each node scores its peers based upon each peer behaviour, and then this scoring limits message transmission only to peers reaching a certain score threshold [19]. Nodes can also publish to unsubscribed topics via fan-out peering. In this unidirectional mechanism, they send their message to 3 randomly picked peers that subscribed to the topic, called fan-out peers, and then redistribute the message to the network [18]. Thus by design, GossipSub is highly scalable, reliable, fast and efficient, resilient and attack-resistant [17].

2.3.4. Libp2p EpiSub

EpiSub is a proximity aware mono-source multicast optimized PubSub protocol [20] implementing epidemic broadcast trees Plumtree protocol [21] (Gossip-based spanning tree construction, tree repair and optimization). HyParView membership protocol [22] manages EpiSub peers. Each node has two different views in this protocol: a small active one that engenders a message-passing overlay. A larger passive one maintains active view network resilience in case of node failure. EpiSub implements GoCast proximity-aware overlay scheme [23] which dynamically maintains near neighbourhood connectivity, i.e. low latency neighbours, by a near degree node confinement method during add and drop connection phases. Libp2p EpiSub comprises two main protocols: the broadcast protocol (publish) that uses lazy multicast tree construction through Plumtree's epidemic broadcast [21]. The second protocol is membership management (subscribe), which maintains active and passive peers lists for a related topic. Eager and lazy active peers are distinguished, eager ones actively disseminating new messages at the edges of the multicast tree, and lazy ones only gossiping about message summaries and maintaining the multicast tree [20]. Multicast tree and peer proximity optimization [23] constantly optimize transmission latency and propagation latency. An implementation of EpiSub is currently under active development in the scope of this project.

3. PROPOSED ALGORITHMS

3.1. Simple distributed cellular automata

3.1.1. Used cellular automata and conventions

The CA uses Conway's Game of Life: two states, grade IV [24], totalistic, Moore's neighborhood driven [25], elementary cellular automaton. This CA will run on a 2-dimensional circular grid. Each cell ran independently and asynchronously, as subscription events trigger its transition function: its own cell for the neighbour publishing version and its eight neighbouring cells for the neighbour subscribing version. A generic PubSub employed functions *pubsub.pub* as publish function and *pubsub.sub* as subscribe function. Publish function takes for argument the name of the topic and the message to publish. Subscribe function takes for argument the name of the topic to subscribe and a callback function that processes received messages. These two generic functions can be adapted to match FloodSub, GossipSub, and EpiSub versions. Each published message reflects the cell state or a start action request. Message can be 0 (dead cell), 1 (alive cell) or 2 (start request). Any cell of the CA can receive a start request, then the triggered cell broadcasts its state to its neighbours and effectively starts the CA. Let $N \geq 3$, the CA will run on a $N \times N$ circular grid, and let C a cell of coordinates $(x, y) \in N \times N$. Thus the pubsub topic of C will be "cell- x - y ". For performance comparison purposes, two DCA versions, a neighbour publishing version (NP) and a neighbour subscribing version (NS), detailed below and then implemented.

3.1.2. Neighbour publishing version

In this version, on a $N \times N$ CA, $N \geq 3$, each cell has one topic subscription, thus N^2 total subscriptions, and at each round, a cell publishes to its eight neighbours as shown in Figure 1(a), thus $8 \times N^2$ publications per round. Algorithm 1 shows a single cell class, instantiating a cell object member of the CA. The initialization parameter *coord* is in the form $(x, y) \in N \times N$, the *state* parameter is either 0 or 1 and the *length* parameter is N . The *neighb* variable is an array of the cell neighbour coordinates, thus an array of length 8 as Moore's neighbourhood [25] is used. Variable *alive_neighb* is the number of alive neighbours for this current round, and *current_neighb* is the number of messages received by its neighbourhood. Variable *subscribe* keeps the pubsub mechanism, processes each event received by updating the cell and broadcasting each cell state update to its neighbourhood.

3.1.3. Neighbour subscribing version

In this version, on a $N \times N$ CA, $N \geq 3$, each cell has 8 topics subscriptions as shown in Figure 1(b) thus $8 \times N^2$ total subscriptions, and at each round a cell publishes to its own topic, thus N^2 publications per round. Algorithm 2 shows a single cell class, instantiating a cell object member of the CA. The initialization parameter *coord* is in the form $(x, y) \in N \times N$, the *state* parameter is either 0 or 1 and the *length* parameter is N . Variable *alive_neighb* is the number of alive neighbours for this current round, and *current_neighb* is the number of messages received by its neighbourhood. Variable *subs* keeps the pubsub mechanism as an array of neighbourhood subscriptions that processes each event received, updates the cell and broadcasts the cell state update to its current topic.

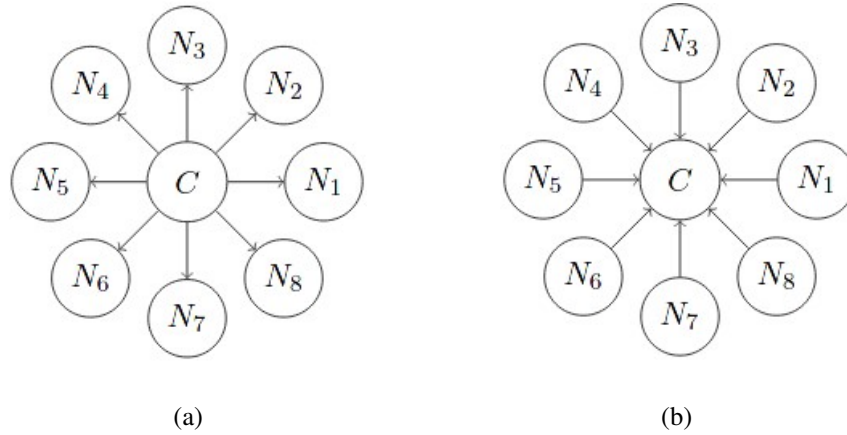


Figure 1. Cellular automata publishing/subscribing schemes for (a) Cell C publishing to neighbours N_i and (b) Cell C subscribing to neighbours N_i

Algorithm 1 Cell object: Neighbour publishing version

```

class CELL_NP(COORD,STATE,LENGTH)
  coord ← coord
  state ← state
  neighb ← neighb_list(coord,length)
  alive_neighb ← 0
  current_neighb ← 0
  subscribe ← cell_subscribe(this,neighb_publish)
end class

```

Algorithm 2 Cell object: Neighbour subscribing version

```

class CELL_NS(COORD,STATE,LENGTH)
  coord ← coord
  state ← state
  alive_neighb ← 0
  current_neighb ← 0
  subs ← neighb_subscribe(this,length,cell_publish)
end class

```

3.1.4. Publish-subscribe functions

PubSub functions of both CA versions used above can be described as: i) On a Neighbour publishing scheme; ii) Each cell uses the publish function to send a message to its neighbours; and iii) The subscribe function listens to its topic. In Algorithm 3, these are *neighb_publish* and *cell_subscribe* functions. On a neighbour subscribing scheme, each cell uses the publish function to update its topic and the subscribe function is used to listen to its neighbours' topics. In Algorithm 3, these are *cell_publish* and *neighb_subscribe* functions. It is essential to mention that the *cell* function parameter is just a reference to the cell object and not the cell object itself.

3.1.5. Message processing function

The message processing function plays the role of an asynchronous CA transition function and is used for both versions. As shown in Algorithm 4, this function takes for argument a cell reference and a pubsub publication callback, and then returns a function that processes a message, starts or updates a cell given its current neighbourhood state according to Conway's game of life rules: any alive cell surrounded by two alive

neighbours lives on the next state, any cell surrounded by three alive neighbours lives on the next state, and in any other case, the cell dies on the next state. The publication callback is called either when a start request is made ($msg = 2$), and in that case, it starts its neighbourhood by broadcasting its current state, or the cell has reached a new round and has updated its state, and in that case, it publishes its new state.

Algorithm 3 Publish and subscribe functions

```

procedure CELL_PUBLISH(cell)
  pubsub.pub(cell_name(cell.coord), cell.state)
end procedure
procedure NEIGHB_PUBLISH(cell)
  for  $i \in cell.neighb$  do
    pubsub.pub(cell_name(cell.neighb[i]), cell.state)
  end for
end procedure
function CELL_SUBSCRIBE(cell, publish)
  sub  $\leftarrow$  pubsub.sub(cell_name(cell.coord))
  sub.on("message", process(cell, publish))
  return sub
end function
function NEIGHB_SUBSCRIBE(cell, length, publish)
  neighb  $\leftarrow$  neighb_list(cell.coord, length)
  sub  $\leftarrow$  Array(8)
  for  $i \in neighb$  do
    name  $\leftarrow$  cell_name(neighb[i].coord)
    cur_neighb  $\leftarrow$  pubsub.sub(name)
    cur_neighb.on("message", process(cell, publish))
    sub[i]  $\leftarrow$  cur_neighb
  end for
  return sub
end function

```

Algorithm 4 Message processing function

```

function PROCESS(cell, publish_callback)
  return
function MESSAGE_PROCESSOR(msg)
  if  $msg = 2$  then
    return publish_callback(cell)
  else
    alive_neighb  $\leftarrow$  alive_neighb + msg
    current_neighb  $\leftarrow$  current_neighb + 1
    if current_neighb = 8 then
      if (alive_neighb = 2 and cell.state = 1) or alive_neighb = 3 then
        cell.state  $\leftarrow$  1
      else
        cell.state  $\leftarrow$  0
      end if
      (alive_neighb, current_neighb)  $\leftarrow$  (0, 0)
      return publish_callback(cell)
    end if
  end if
end function
end function

```

3.1.6. Miscellaneous functions

Two functions remain necessary to complete the CA, they are used in both versions and are described in Algorithm 5. Function *cell_name* returns the topic name of a given cell, and *neighb_list* function returns an array of neighbours coordinates for a given cell of coordinates *coord* and a given *length* = *N* on a $N \times N$, $N \geq 3$, circular grid.

Algorithm 5 Miscellaneous functions

```

function CELL_NAME(coord)
  return "cell - ".coord[0]. - ".coord[1]
end function
function NEIHB_LIST(coord, len)
  neighb  $\leftarrow$  Array(0)
  (x, y)  $\leftarrow$  coord
   $\Delta \leftarrow [-1, 0, 1]$ 
  for ( $\alpha, \beta$ )  $\in \Delta \times \Delta$  do
    if  $\Delta[\alpha] \neq 0$  or  $\Delta[\beta] \neq 0$  then
       $\mu \leftarrow (len + x + \Delta[\alpha]) \bmod len$ 
       $\nu \leftarrow (len + y + \Delta[\beta]) \bmod len$ 
      neighb.push(( $\mu, \nu$ ))
    end if
  end for
  return neighb
end function

```

3.1.7. CA initialization and known limitations

The CA can be initialized by instantiating the $N \times N$ cells and then send a message $msg=2$ to any topic "cell- x - y " where $(x, y) \in N \times N$. The triggered cell then propagates its current state and effectively starts the whole CA in an asynchronous domino effect. This straightforward implementation has known limitations, as a cell does not keep track of its current round and thus does not know if received messages are related to its current or next round. The following CA implementation tackle this problem.

3.2. Decentralized swarms of distributed CA**3.2.1. Description and conventions**

For large scale purposes, the asynchronous distributed CA divided into swarms of autonomous smaller CA with inner-swarm and inter-swarm communication capabilities. Communication is necessary inside the swarm and between the cells located at the edges of two distinct swarms. This will be achieved through local cell-to-cell and global swarm-to-swarm communication channels as shown in Figure 2. Each cell has to keep track of its current round, and process its neighbourhood messages according to its round number. At round 0, each cell is unaware of its neighbours' swarm. Each cell identifies itself to its neighbourhood during this discovery phase through direct messaging as in the NP DCA version previously described. From round 1 begins the processing phase, where each cell is aware of the corresponding swarms of its neighbourhood, and communication is provided through inner-swarm and inter-swarm broadcast.

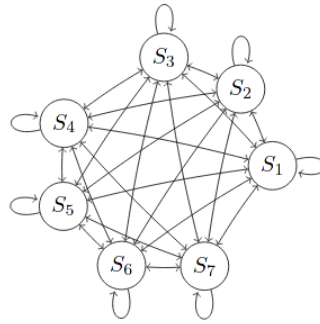


Figure 2. Swarms S_i of distributed CA communicating

3.2.2. Cell and swarm objects

Algorithm 6 describes a single cell class, instantiating a cell object member of a swarm. The initialization parameter *coord* is in the form $(x, y) \in N \times N$, *state* parameter is either 0 or 1, *swarm* parameter is a reference to the related swarm of the cell and *length* parameter is N. Variable *neighb* is an array of 8 neighbouring cells where each item has its swarm id, cell coordinates, and its two last state rounds stored. Variable

subscribe keeps the pubsub mechanism at a local level, processing each event received, updating the cell and broadcasting cell state updates to its neighbourhood. Algorithm 7 describes a single swarm class, instantiating a swarm object member of the CA. The initialization parameter *swarm_id* represents the id of the swarm, and *cell_array* parameter assigned to *cells* variable is the array of cell objects from the swarm. Variable *subscribe* keeps the pubsub mechanism at both local and global levels, processing and dispatching each event received depending on its origin and destination.

Algorithm 6 Cell object: Swarm version

```

class CELL(COORD,STATE,SWARM,LENGTH)
  coord ← coord
  state ← state
  swarm_id ← swarm_id
  round ← 0
  neighb ← neighb_list(coord,length)
  subscribe ← cell_subscribe(this,swarm)
end class

```

Algorithm 7 Swarm object

```

class SWARM(SWARM_ID,CELL_ARRAY)
  id ← swarm_id
  cells ← cell_array
  subscribe ← swarm_subscribe(this)
end class

```

3.2.3. Publish and subscribe functions

PubSub functions enable local and inter-swarm communication. On a local level, each cell uses the publish function to send a message to its swarm, and the subscribe function listens and processes messages from its topic. In Algorithm 8, these are the functions *cell_publish* and *cell_subscribe*. On a global level, each swarm uses the publish function to dispatch a received message, either to its cells or to another swarm, and the subscribe function listens and processes messages from its topic. In Algorithm 8, these are the functions *swarm_publish* and *swarm_subscribe*. Functions *pubsub.sub* and *pubsub.sub* are generic enough to implement either FloodSub, GossipSub or EpiSub PubSub models.

Algorithm 8 Publish and subscribe functions

```

function SWARM_SUBSCRIBE(swarm)
  sub ← pubsub.sub(swarm_name(swarm.id))
  sub.on("message",swarm_process(swarm))
  return sub
end function
procedure SWARM_PUBLISH(swarm,msg)
  if swarm.id = msg[0] then                                     ▷ process local message
    (swarm_process(swarm))(msg)
  else                                                         ▷ dispatch to another swarm
    pubsub.pub(swarm_name(msg[0]),msg)
  end if
end procedure
function CELL_SUBSCRIBE(cell,swarm)
  sub ← pubsub.sub(cell_name(cell.coord))
  sub.on("message",cell_process(cell,swarm))
  return sub
end function
procedure CELL_PUBLISH(msg)
  pubsub.pub(cell_name(msg[1]),msg)
end procedure

```

3.2.4. Cell message broadcast

The cell message broadcast procedure described in Algorithm 9 sends a message to all neighbours of a given cell through swarm_publish procedure.

Algorithm 9 Cell message broadcast

```

procedure CELL_MSG_BROADCAST(cell,swarm)
  for  $i \in \text{cell.neighb}$  do
     $\text{cur\_neighb} \leftarrow \text{cell.neighb}[i]$ 
     $\text{msg} \leftarrow (\text{cur\_neighb.swarm\_id}, \text{cur\_neighb.coord}, \text{cell.coord}, \text{cell.round}, \text{cell.state})$ 
     $\text{swarm\_publish}(\text{swarm}, \text{msg})$ 
  end for
end procedure

```

3.2.5. Swarm processing function

The swarm processing function plays the role of an asynchronous message router. As shown in Algorithm 10, this function takes for argument the current swarm reference. It returns a function that processes each message as follows: if the message is heading to an inner cell, the target cell processes it directly. If the message is heading to a known swarm, the pubsub swarm_publish function will send it. In the initialization phase, the swarm destination is unknown. Thus the message is directly dispatched to the corresponding cell via pubsub cell_publish function.

Algorithm 10 Swarm processing function

```

function SWARM_PROCESS(swarm)
  return
  function MESSAGE_ROUTER(msg)
     $\text{coord} \leftarrow \text{msg}[1]$ 
     $\text{cell\_id} \leftarrow \text{find\_cell\_id}(\text{swarm.cells}, \text{coord})$ 
    if  $\text{cell\_id} \geq 0$  then
      return  $(\text{cell\_process}(\text{swarm.cells}[\text{cell\_id}], \text{swarm}))(msg)$ 
      ▷ Incoming message
    else if  $\text{msg}[0] \geq 0$  then
      return  $\text{swarm\_publish}(\text{swarm}, \text{msg})$ 
      ▷ Known swarm publish
    else
      return  $\text{cell\_publish}(msg)$ 
      ▷ Unknown swarm publish
    end if
  end function
end function

```

3.2.6. Cell processing function

The cell processing function plays the role of an asynchronous transition function. As shown in Algorithm 11, this function takes for argument a cell reference and a pubsub publication callback and returns a function that processes a message, starts or updates a cell depending on the message received. If the received swarm id is -1, the cell identifies itself on the network, broadcasting its swarm id, coordinates and current state to its neighbourhood. In other cases, the received message is processed in the neighbour array. Suppose the cell received enough neighbours for its round (processed via *alive_n* function). In that case, Conway's game of life rules are applied, the neighbour state array is reset for this round (via *reset_neighb* procedure), and the cell message broadcast procedure sends the next state to its neighbours.

Algorithm 11 Cell processing function

```

function CELL_PROCESS(cell,swarm)
  return
  function MESSAGE_PROCESSOR(msg)
    (swarm_id,coord,target,round,state)  $\leftarrow$  msg
    if swarm_id = -1 then                                     ▷ Start the CA
      cell_msg_broadcast(cell, swarm)
      return true
    end if
    neighb_id  $\leftarrow$  coord_to_id(cell.neighb,target)
    cell.neighb[neighb_id].swarm_id  $\leftarrow$  swarm_id
    cell.neighb[neighb_id].s[round mod 2]  $\leftarrow$  state
    (alive,total)  $\leftarrow$  alive_n(cell.neighb,cell.round)
    if total = 8 then                                         ▷ All neighbours received
      if (alive = 2 and cell.state = 1) or alive = 3 then
        cell.state  $\leftarrow$  1
      else
        cell.state  $\leftarrow$  0
      end if
      cell.round  $\leftarrow$  cell.round + 1
      reset_neighb(cell)
      cell_msg_broadcast(cell, swarm)
      return true
    end if
    return false
  end function
end function

```

4. PRELIMINARY RESULTS - PERFORMANCE COMPARISONS**4.1. Implementation and experimentation testbed**

DCA and DSDCA algorithms were implemented in the NodeJS framework and interfaced with a tweaked version of IPFS, compatible with Linux x64 and OS X platforms. Source code is freely available on GitHub [26]. The following performance measures were carried out using Grid'5000 testbed [27], a large-scale testbed for experiment-driven research supported by a scientific interest group (GIS) hosted by Inria, including CNRS, RENATER, and several French Universities as well as other organizations. Only IPFS bandwidth was measured during preliminary experiments, but inconsistencies between IPFS and network bandwidth, including TCP/IP overhead, were observed. It was then decided that total network bandwidth usage, including TCP/IP overhead, was the most relevant resource to monitor.

4.2. DCA NP/NS Grid'5000 experimentation protocol

Experimentations with DCA NP and NS versions were carried out on various clusters from 5 different sites (Lyon, Lille, Nancy, Nantes, Grenoble). Each experiment was launched for 100 rounds on five machines in parallel, and network bandwidth was measured with vnstat. Various settings were used, such as the number of nodes (from 10 to 226), FloodSub and GossipSub router protocols, and NP / NS DCA versions. At the end of each experiment, the total network bandwidth used for IPFS nodes bootstrapping and CA processing was measured, and the average network bandwidth per node was processed. Each DCA ran on individual machines, and network bandwidth was measured through vnstat software in interactive mode, monitoring each machine's loopback (lo).

4.3. Distributed cellular automata performance

DCA performance was measured during 250 experiments with several nodes ranging from 10 to 226. The focus is kept on results for experiments with more than 50 nodes. As shown in Figures 3(a) and 3(b), there is no meaningful difference in network bandwidth between both PubSub protocols when comparing NP or NS implementations, 0.1% for NP version and 0.3% for NS version on average. Contrarily, it is much relevant to compare NP and NS implementations running with the same PubSub protocol, as shown in Figure 4(a) and 4(b).

On average, there is a 24.7% bandwidth difference for Floodsub protocol and a 24.3% bandwidth difference for GossipSub protocol. In any case, and contrarily to what was supposed theoretically, the NP version seems to be the most bandwidth-efficient algorithm, even independently from FloodSub or GossipSub protocol.

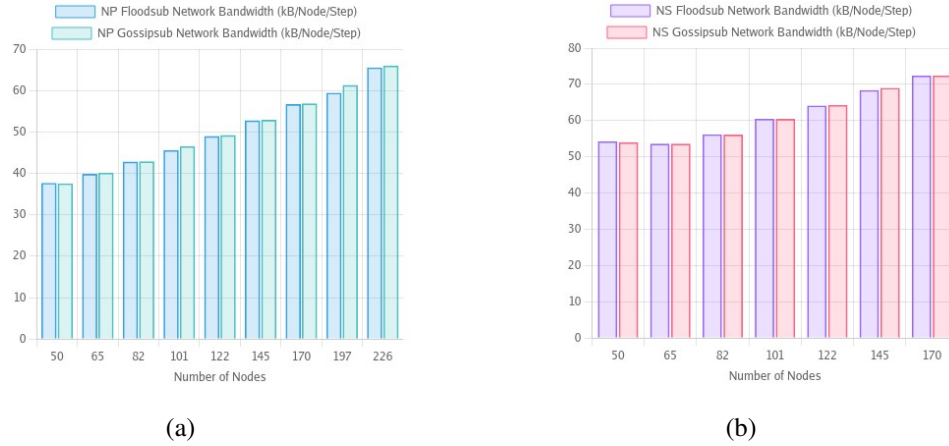


Figure 3. DCA: Floodsub vs Gossipsub network bandwidth for (a) NP version in kilobit per node per step and (b) NS version in kilobit per node per step

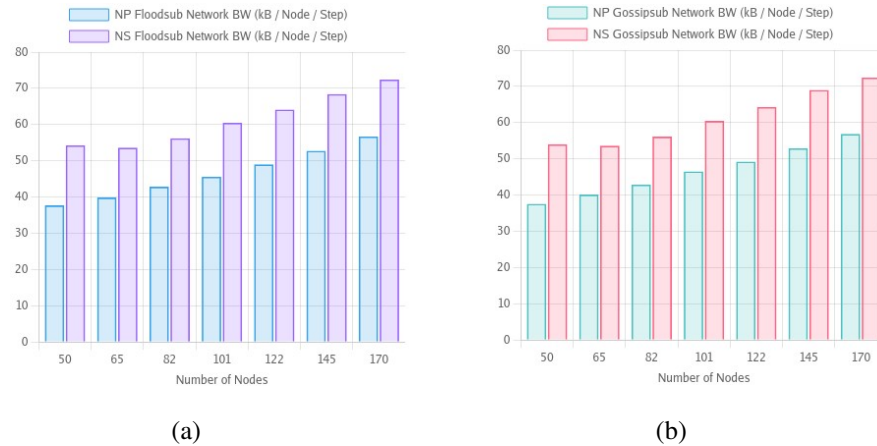


Figure 4. DCA: Floodsub and Gossipsub network bandwidth for (a) NP Floodsub vs NS Floodsub in kb/node/step and (b) NP Gossipsub vs NS Gossipsub in kb/node/step

4.4. DSDCA Grid'5000 experimentation protocol

Experimentations for Decentralized Swarms of DCA were carried out across five sites with machines from various clusters with 10 GBps connectivity (Nova-Lyon, Chiclet-Lille, Gros-Nancy, Ecotype-Nantes, Dahu-Grenoble). Each swarm was running on a specific site, and network bandwidth was measured with vnstat software. Nova was used as the IPFS private network and swarm bootstrapping node, Chiclet, Gros, Ecotype, and Dahu as Swarms of DCA nodes. Various settings were tried, such as the number of swarms (3 and 4), nodes (from 13 to 404), and FloodSub and GossipSub router protocols. Each experiment was run five times for 1000 rounds. At the end of each experiment, the total network bandwidth used between sites was measured, and the average network bandwidth per node was processed.

4.5. Decentralized swarms of DCA performance

During 250 different experiments, DSDCA implementation performance was measured with several nodes ranging from 13 to 404. Network bandwidth measures for the bootstrap phase (IPFS private network and node bootstrapping) was dissociated from the DSDCA processing phase. Concerning the bootstrap phase in three swarms and four swarms experiments, there is a substantial network bandwidth difference between FloodSub and GossipSub versions, as shown in Figures 5(a) and 5(b). On average, there is a 31.3% bandwidth difference for FloodSub vs GossipSub protocol on three swarms bootstrap phases, and 11.7% bandwidth difference for FloodSub vs GossipSub protocol on four swarms bootstrap phases.

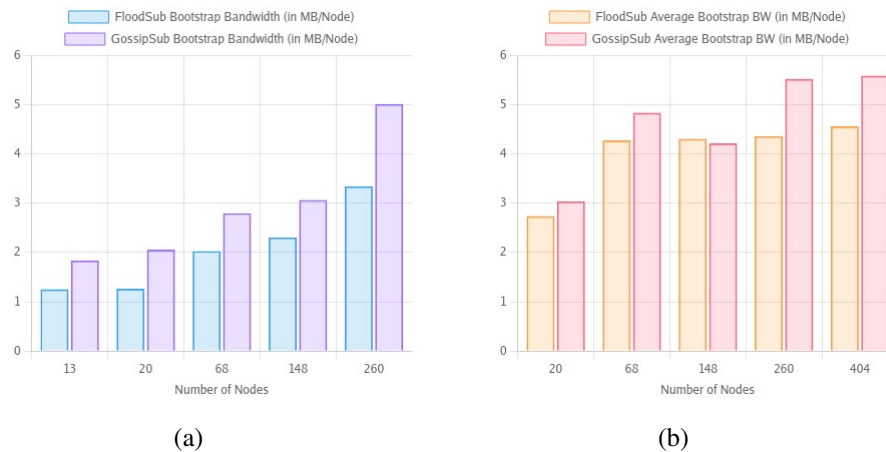


Figure 5. DSDCA: Floodsub vs Gossipsub bootstrap bandwidth for (a) 3 Swarms DSDCA in Mb/node and (b) 4 Swarms DSDCA in Mb/node

Regarding the DSDCA processing phase in three swarms and four swarms experiments, network bandwidth variations between FloodSub and GossipSub versions are pretty significant and informative, as shown in Figures 6(a) and 6(b). On average, there is a 55,45% bandwidth difference for FloodSub vs GossipSub protocol on three swarms bootstrap phases, and 34.16% bandwidth difference for FloodSub vs GossipSub protocol on four swarms bootstrap phases. In any case, as a preliminary result, and contrarily to what was initially supposed, FloodSub is observed to be the most bandwidth-efficient protocol for the DSDCA algorithm, independently from the number of nodes and of swarms.

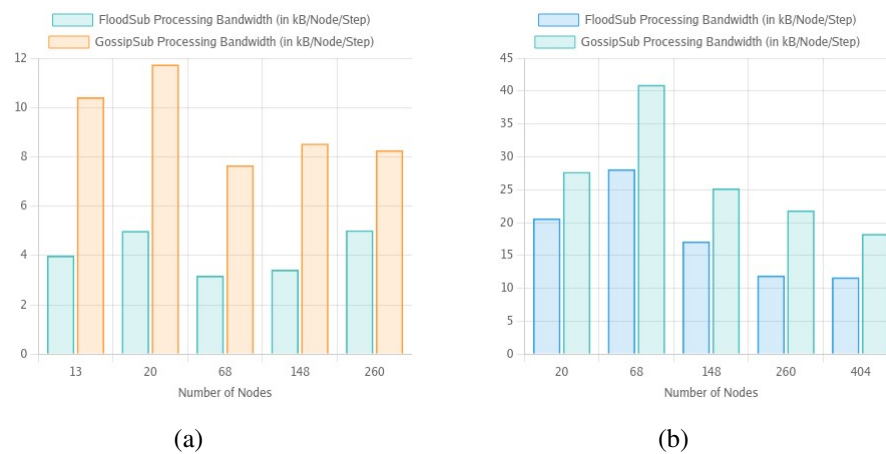


Figure 6. DSDCA: Floodsub vs Gossipsub processing bandwidth for (a) 3 Swarms DSDCA in kb/node/step and (b) 4 Swarms DSDCA in kb/node/step

4.6. A few statistics about Grid'5000 usage

This experiment used 4,701 CPU core hours on 47 machines across Lille, Grenoble, Lyon, Nancy, and Nantes datacenters. The total network bandwidth used across Grid'5000 datacenters is estimated to be 137.475 TB for NP/NS DCA versions experiments and 148.396 TB for decentralized swarms of DCA experiments. About 500 different experiments were run during two weeks, about 43,000 IPFS nodes were fired up, and 30,000,000 rounds of CA were computed.

5. CONCLUSION AND FURTHER WORKS

This article introduced an experimental implementation of simple asynchronous distributed cellular automata and decentralized swarms of asynchronous distributed cellular automata driven by IPFS and Libp2p PubSub. PubSub functions genericity provides the possibility to switch between FloodSub, GossipSub and EpiSub protocols. Experiments were carried out on Grid'5000 testbed for either NP/NS DCA and 3/4 swarms DSDCA. Preliminary results showed the sustainability of NP version over NS DCA and FloodSub protocol over GossipSub for DCA and DSDCA algorithms.

Further experiments will be conducted with a larger number of Swarms and Nodes, and a comparison of protocol performance will be made with the EpiSub protocol. EpiSub implementation over Libp2p/IPFS is currently under active development, as an automated DSDCA experiment launcher and bandwidth monitoring over Grid'5000, and a real-time asynchronous distributed cellular automata visualization system. This work is prior to implementing a large-scale decentralized epidemic propagation modelling and prediction system based upon asynchronous distributed cellular automata applied to the current SARS-CoV-2 (COVID-19) epidemic. This system will use a hybrid adapted version of SIR models driven by DSDCA and fed by worldwide data, estimating population dynamics on various factors such as mobility, infection rates, vaccine rates, recoveries and deaths. In this DSDCA system, individual nodes will simulate population dynamics in cities, swarms for regions and upper-level swarms for countries. Although cellular automata can be computed faster in a centralized manner, decentralization will enable to free from memory and processing power limits induced by centralization in the context of such a very large scale simulation application.

ACKNOWLEDGEMENT

Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER, and several universities as well as other organizations (see <https://www.grid5000.fr>). I want to express my gratitude and thanks to Pierre Neyron from CNRS, which granted me Grid'5000 open access to run DCA and DSDCA experiments across 5 French datacenters. I want to thank especially George Polyzos from AUEB, Nuno Santos from INESC-ID and Jorge Soares from Protocol Labs, who reviewed this paper at a very early stage and whose feedbacks motivated me to elaborate and dig further. I would also like to thank the French Government for awarding social grants, providing me with the financial means to complete this self-funded, non-profit and independent research.





REFERENCES

- [1] J. Santos, N. Santos, and D. Dias, "Censorship-resistant web annotations based on ethereum and IPFS," in *Proceedings of the 35th Annual ACM Symposium on Applied Computing (SAC '20)*. Association for Computing Machinery, 2020, pp. 2211–2213, doi: 10.1145/3341105.3374049.
- [2] A. J. Collados-Lara, E. Pardo-Igúzquiza, and D. Pulido-Velazquez, "A distributed cellular automata model to simulate potential future impacts of climate change on snow cover area," *Advances in Water Resources*, vol.124, pp. 106–119, 2019, doi: 10.1016/j.advwatres.2018.12.010.
- [3] J. Benet, "IPFS - content addressed, versioned, P2P file system," *arXiv Networking and Internet Architecture*, arXiv:1407.3561, 2014.
- [4] S. Ulam, "On some mathematical properties connected with patterns of growth of figures," *Proceedings of Symposia on Applied Mathematics*, vol.14, pp. 215–224, 1962.
- [5] J. Von Neumann and A. W. Burks, *Theory of self-reproducing automata*, 1st ed., Illinois, USA: University of Illinois Press, 1966.
- [6] V. Christianto, V. Krasnoholovets, and F. Smarandache, "Cellular automata representation of submicroscopic physics," *Prespacetime Journal*, vol. 10, no. 8, pp. 1024–1036, Dec. 2019.
- [7] P.M.A. Sloot, J.A. Kaandorp, A.G. Hoekstra, and B. Overeinder, "Distributed cellular automata: large scale simulation of natural phenomena," *Computer Physics Communications*, Jan. 2001.
- [8] H. Huang, J. Lin, B. Zheng, Z. Zheng, and J. Bian, "When blockchain meets distributed file systems: an overview, challenges, and open issues," *IEEE Access*, vol. 8, pp. 50574–50586, 2020, doi: 10.1109/ACCESS.2020.2979881.

- [9] D. Dias and J. Benet, "Distributed web applications with IPFS, tutorial," in *Web Engineering. ICWE 2016. Lecture Notes in Computer Science*, vol. 9671, A. Bozzon, P. Cudre-Maroux, and C. Pautasso, Eds., 2016, pp. 616–619.
- [10] S. S. Hasan, N. H. Sultan, and F. A. Barbhuiya, "Cloud data provenance using IPFS and blockchain technology," in *Proceedings of the Seventh International Workshop on Security in Cloud Computing (SCC '19)*, 2019, pp. 5–12, doi: 10.1145/3327962.3331457.
- [11] Protocol Labs, "Filecoin: a decentralized storage network," *Protocol Labs Research*, July 2017. Available: <https://research.protocol.ai/publications/filecoin-a-decentralized-storage-network> (accessed Sep. 12, 2020).
- [12] S. He, Y. Lu, Q. Tang, G. Wang, and C. Qishi Wu, "Peer-to-peer content delivery via blockchain," *arXiv Cryptography and Security*, arXiv:2102.04685, 2021.
- [13] J. P. de Araujo, "A communication-efficient causal broadcast publish/subscribe system," Ph.D. dissertation, Departement of Informatique, Télécommunications et Électronique, Sorbonne Université, Paris, France, pp. 19–21, 2019.
- [14] V. Santos, "Js-libp2p-floodsub", *Libp2p*, Dec. 2020. Available: <https://github.com/libp2p/js-libp2p-floodsub> (accessed Feb. 14, 2021).
- [15] D. Dias, "PubSub at Scale," *Protocol/ResNetLab Labs*, Sep. 2020. Available: https://github.com/protocol/ResNetLab/blob/master/OPEN_PROBLEMS/PUBSUB_AT_SCALE.md (accessed Feb. 20, 2021).
- [16] R. Baldoni, R. Beraldi, V. Quema, L. Querzoni, and S. Tucci-Piergiovanni, "TERA: topic-based event routing for peer-to-peer architectures," *DEBS '07: Proceedings of the First ACM International Conference on Distributed Event-Based Systems*, Jan. 2007, pp. 2–13, doi:10.1145/1266894.1266898.
- [17] D. Vyzovitis, Y. Napor, D. McCormick, D. Dias, and Y. Psaras, "GossipSub: attack-resilient message propagation in the Filecoin and ETH2.0 networks," in *Proceedings of Protocol Labs TechRep (PL-TechRep-2020-002)*, arXiv:2007.02754, 2020.
- [18] "Publish / Subscribe," *Libp2p*. Available: <https://docs.libp2p.io/concepts/publish-subscribe> (accessed: Feb 19, 2021).
- [19] D. Vyzovitis, "Gossipsub v1.1: Security extensions to improve on attack resilience and bootstrapping," *Libp2p*, Dec. 2020. Available: <https://github.com/libp2p/specs/blob/master/pubsub/gossipsub/gossipsub-v1.1.md> (accessed Feb 2, 2021).
- [20] D. Vyzovitis, "Episub: Proximity Aware Epidemic PubSub for libp2p," *Libp2p*, June 2019. Available: <https://github.com/libp2p/specs/blob/master/pubsub/gossipsub/episub.md> (accessed Feb. 14, 2021).
- [21] J. Leita, J. Pereira, and L. Rodrigues, "Epidemic broadcast trees," *2007 26th IEEE International Symposium on Reliable Distributed Systems (SRDS 2007)*, 2007, pp. 301–310, doi: 10.1109/SRDS.2007.27.
- [22] J. Leita, J. Pereira, and L. Rodrigues, "HyParView: a membership protocol for reliable gossip-based broadcast," *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*, 2007, pp. 419–429, doi: 10.1109/DSN.2007.56.
- [23] C. Tang, R. N. Chang, and C. Ward, "GoCast: gossip-enhanced overlay multicast for fast and dependable group communication," *2005 International Conference on Dependable Systems and Networks (DSN'05)*, 2005, pp. 140–149, doi: 10.1109/DSN.2005.52.
- [24] S. Wolfram, *A new kind of science*, 1st ed., Champaign, IL, USA: Wolfram Media, 2002, pp. 231–249.
- [25] T. Toffoli and N. Margolus, *Cellular automata machines: a new environment for modeling*, Cambridge, MA, USA: The MIT Press, 1987, doi: 10.7551/mitpress/1763.001.0001.
- [26] V. Manuceau, "About decentralized swarms of Aasynchronous distributed cellular automata using IPFS," *GitHub*, July 2021. Available: https://github.com/vincent-manuceau/Decentralized_Cellular_Automata_over_IPFS (accessed Aug. 30, 2021).
- [27] F. Desprez and L. Nussbaum, "The data-centers facet of SILECS (A.K.A. Grid'5000)" *SILECS/Datacenters - Grid'5000*, April 2019. Available: <https://www.grid5000.fr/mediawiki/images/Grid5000.pdf> (accessed Aug. 30, 2021).

BIOGRAPHY OF AUTHOR



Vincent Manuceau     is an independent researcher with main interests in cellular automata, inter-planetary file system (IPFS) and recursive internetwork architecture (RINA). Currently Head of Creole Shop company, the leading Caribbean food and beverages online grocery store with 24h/48h shipping worldwide (<https://www.creole-shop.fr/en>), he develops innovative technologies with direct applications to the company, such as full automation of the logistics chain. He does fundamental research on his spare time, and writes computer science and mathematics courses, he also is reviewer for international journal of informatics and communication technology (IAES II-ICT) since 2021. Vincent is always open to knowledge sharing and teaching. Further info on his homepage: <http://vincent.manuceau.net>. He can be contacted at email: vincent@manuceau.net.