Pleiades: Distributed Structural Invariants at Scale

Simon Bouget, Yérom-David Bromberg, Adrien Luxey, François Taïani

Pleiades: Distributed Structural Invariants at Scale

Simon Bouget, Yérom-David Bromberg, Adrien Luxey, François Taiani
Univ Rennes, CNRS, Inria, IRISA
F-35000 Rennes, France
Email: {francois.taiani,simon.bouget,david.bromberg,adrien.luxey}@irisa.fr

***Abstract*—Modern large scale distributed systems increasingly espouse sophisticated distributed architectures characterized by complex distributed structural invariants. Unfortunately, maintaining these structural invariants at scale is time consuming and error prone, as developers must take into account asynchronous failures, loosely coordinated sub-systems and network delays.**

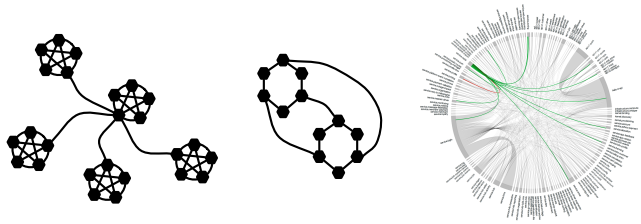
To address this problem, we propose PLEIADES, a new framework to construct and enforce large-scale distributed structural invariants under aggressive conditions. PLEIADES combines the resilience of self-organizing overlays, with the expressiveness of an assembly-based design strategy. The result is a highly survivable framework that is able to dynamically maintain arbitrary complex distributed structures under aggressive crash failures. Our evaluation shows in particular that PLEIADES is able to restore the overall structure of a 25,600 node system in less than 11 asynchronous rounds after half of the nodes have crashed.

I. INTRODUCTION

Modern distributed architectures are becoming increasing large and complex. They typically bring together independently developed sub-systems (e.g. for storage, batch processing, streaming, application logic, logging, caching) into large, geo-distributed and heterogeneous architectures [16]. These complex architectures often require structural invariants to be maintained in order to ensure the correct functioning of the overall system, i.e. regarding the number of nodes, their connections, and the system's overall topology.

For instance, *MongoDB* [26], a popular document-oriented no-sql database, must maintain a star topology between sets of nodes organized in cliques (Figure 1a). Similarly the cross-datacenter replication feature of *Riak* [33], a production-level key-value datastore, requires multiple rings to maintain connections across geo-distributed datacenters (Figure 1b). Another example are large scale distributed systems such as for example *Ceph* [38] or *Glusterfs*[1] that rely on a hierarchical cluster of nodes to provide reliable data distribution.

These systems only provide a single service (storage), and must be combined with other sub-systems to construct actual applications. Complete distributed applications thus present even more complex architectures which may for instance combine peer-to-peer elements and edge-servers, with datacenter-hosted machines in a wide range of topologies [10], [16], [41]. This trend is compounded by the fact that, in the wake of Netflix, a growing number of applications are today adopting a microservice architecture. Such applications involve thousands of services interconnected to form complex intertwining structures (Figure 1c).



(a) MongoDB (b) Riak (c) Microservices

Figure 1: Modern distributed systems must maintain increasingly complex structural invariants

Maintaining such structural invariants at scale is particularly challenging as it requires maintaining systemic properties in spite of continuous asynchronous failures, loosely coordinated sub-systems, and network delays. As a result, current distributed management frameworks such as *Borg* [35], *Mesos* [15] or *Kubernetes* [7] have so far limited themselves to basic structural properties mainly based on cardinality (e.g. maintaining X instances of the same container, or same group of containers). This is problematic for developers who must increasingly enforce sophisticated invariants as distributed systems continue to grow in scope and size.

In this paper we propose PLEIADES, a new framework to construct and enforce *large-scale distributed structural invariants* in a way that is *autonomous* and *resilient*. Our approach leverages the power of self-stabilizing overlays [2], [37], [36] for resilience, and assembly-based modularity [6], [32] for expressiveness into a seamless framework. The result is a framework that can maintain at runtime an arbitrary number of distributed structures under aggressive conditions, including catastrophic failures.

More precisely, we make the following contributions:

- We introduce PLEIADES, an assembly-based topology programming framework that harnesses the autonomous properties of self-organizing overlays to provide structural distributed invariants that are both scalable and highly resilient.
- Through the design of PLEIADES, we demonstrate how multiple self-stabilizing protocols can be combined to produce a sophisticated self-organizing behavior that is both modular and scalable.

[1] https://www.gluster.org

- We present an extensive evaluation of our framework that demonstrates its benefits in terms of expressiveness, efficiency, low overhead, robustness and dynamic reconfiguation. We show for instance how a system with 25,600 nodes organized in a ring of rings is able to reform its overall structure after 50% of the nodes have crashed in 11 rounds, while consuming less than 2kB of communication per node and per round.

The remainder of the paper is organized as follows: we first motivate our work and present some background information (Section II). We then present the design and workings of PLEIADES (Section III), before moving on to its evaluation (Section IV). We discuss related work (Section V) and conclude (Section VI).

II. PROBLEM, VISION, & BACKGROUND

A. Problem and vision

A growing number of distributed systems rely on complex deployment topologies to provide their services. At the level of individual services, *Scatter* [13] for instance constructs a ring of cliques in which each individual clique executes a Paxos instance, resulting in a scalable and resilient key-value store with a high level of consistency. In the same vein, *MongoDB*—a popular document oriented no-sql database—maintains several *replica sets*, a clique of nodes using a leader-election algorithm to implement a master-slave replication scheme, which communicate with app servers following a star topology [26]. *Riak*, a production level key-value datastore derived from Amazon Dynamo, offers a cross-datacenter replication service that connects several *sink* clusters around a *source* cluster in a star topology. Each Riak cluster is deployed in a ring topology, and the source cluster uses special nodes, known as *fullsync coordinators* to handle the replication to each sink [33]. The above systems only provide individual services, and must be combined with other sub-systems to provide a full-fledged application, leading to increasingly complex distributed architectures. This trend has been popularized with the massive adoption of microservices these last years as witnessed by industry leaders like Netflix, Amazon, Twitter, Airbnb, etc.. From their loosely couple nature, thousands of microservices can be composed and structured [34], [25]. However, if the maintenance of each individual microservice has been simplified, it is not the case of the overall microservices ecosystem that becomes more complex.

Maintaining a complex distributed architecture in a large-scale system is particularly challenging. Such architectures typically rely on systemic invariants that degrade rapidly in the face of failures and delays and must be monitored and repaired continuously. Unfortunately, their maintenance requires coordination and distributed knowledge sharing, which are difficult to implement by practitioners without appropriate support.

In small systems, practitioners often overcome this challenge by intervening manually to keep systems running. As systems grow, however, their developers must increasingly rely on ad hoc mechanisms, developed specifically for particular uses cases, which typically leverage simpler coordination mechanisms such as ZooKeeper or Etcd and elaborate on them to maintain more complex invariants. This approach is highly problematic, as it delegates a core aspect a system's correctness to costly, cumbersome, and error-prone practices.

Preliminary solutions to address the structural maintenance of large-scale systems can be found in modern deployment automation tools such as *Borg* [35], *Kubernetes* [7], *Aurora* [2] or *Mesos* [15], and in self-organizing overlays. Unfortunately both types of solution fall short of the resilience needs of large-scale structural invariants. Deployment automation tools have so far only focused on basic cardinality invariants (e.g. in Kubernetes keeping the number of "pods"—a container groupings—that are instantiated in a replica set at a given level), while self-configuring overlays are optimized for simple topologies, that are only suitable for very basic deployments (e.g. a ring, a tree, a star).

B. Self-organizing overlays

Self-organizing overlays [17], [37] are a family of decentralized protocols that are able to autonomously organize a large number of nodes into a predefined topology (e.g. a torus, a ring). Self-organizing overlays are self-healing, and can with appropriate extension, conserve their overall shape even in the face of catastrophic failures [5]. The scalability and robustness of these solutions have made them particularly well adapted to large scale self-organizing systems such as decentralized social networks [24], [2], news recommendation engines [1], and peer-to-peer storage systems [8].

Self-organizing overlays such as T-Man [17] or Vicinity [37] are unfortunately *monolithic* in the sense that they rely on a single user-defined distance function to connect nodes into a target structure. Simple topologies such as ring or torus are easy to realize in this model, but more complex combinations, such as a star of cliques, are more problematic. Self-organizing overlays lack in particular the ability to incrementally describe complex structures, and do not lend themselves as a result to the enforcement of complex structural invariants.

C. Key challenges and roadmap

In this paper, we take a somewhat extreme stance, and argue that distributed structural invariants in modern large-scale distributed systems should be enforced through a *generic*, *systematic* and *survivable* framework (i.e. that is able to withstand catastrophic failures). More precisely, we propose PLEIADES, a framework in which complex structural invariants can be expressed as an *assemblage* of simple shapes, and *autonomously maintained* at runtime in spite of failures, including catastrophic events. PLEIADES harnesses the expressiveness of *assembly-based modularity* and the resilience of basic *self-organizing overlays* to provide self-healing capability for complex distributed systems at scale.

To avoid any single point of failure, we rely on a *fully decentralized design*. Decentralization brings a number of crucial

[2] aurora.apache.org

benefits in terms of fault-tolerance and survivability, but also important challenges in terms of coordination: the lack of any central coordination point makes it hard to enforce system-wide properties, while the scale which we target (several thousands of nodes) renders typical deterministic agreement protocols difficult to implement efficiently.

In our model, a structural invariant is expressed as a combination of invididual *basic shapes* (ring, grid, stars) which are then connected together to describe the constraints the overall system must obey. The creation and maintenance of basic shapes, of their connecting points, and of their connection are maintained through a number of *continuous self-stabilizing protocols*. These individual protocols interact with one-another to deliver the system's overall survivability.

The protocols making up PLEIADES must resolve a number of key challenges in a fully decentralized manner: (i) they must allocate nodes to "system-level" shapes, (ii) construct individual basic shapes, (iii) bootstrap identification and communications between these shapes, (iv) and realize and maintain the *dynamic bindings* that connect individual shapes according to the developer's intent. In the remaining of this paper, we present these different mechanisms and how they are combined to form PLEIADES.

III. THE PLEIADES FRAMEWORK

A. System model and overall organization

We assume that the target system executes on N nodes that communicate through message passing (e.g. using the TCP/IP stack). The overall organization of a node executing PLEIADES is shown in Figure 2. Each node possesses a copy of the system's overall configuration file (shown on the right side of the figure) which describes *(i)* which basic shapes should be instantiated, and *(ii)* how these shapes should be connected. For brevity's sake, we do not discuss how this configuration file is disseminated to every nodes: this step could rely on a gossip broadcast [20], or, in a cloud infrastructure, each node could retrieve the configuration from its original VM image. Because PLEIADES is self-stabilizing, nodes may receive this configuration at different points in time without impacting the system's eventual convergence.

Starting from this configuration file, PLEIADES constructs and enforces the corresponding structural invariant (in Figure 2, two rings connected through two links) thanks to six self-stabilizing and fully decentralized protocols (shown as rectangles in the figure). These six protocols fall in three categories: the three bottom protocols (*Global RPS*, *Same Shape*, and *Remote Shapes*) are membership protocols (denoted by the symbol ⁘), i.e. helper protocols dedicated to locate and sample nodes and shapes. The *Shape Building* protocol (symbol ○) in the middle of the figure constructs individual shapes, while the top two protocols (*Port Selection*, and *Port Connection*) realize the connection between individual shapes (shown with the symbol ▪▪).

These six protocols execute in a fully decentralized manner, without resorting to any centralized entities, a key property regarding the scalability and resilience of our approach. Each

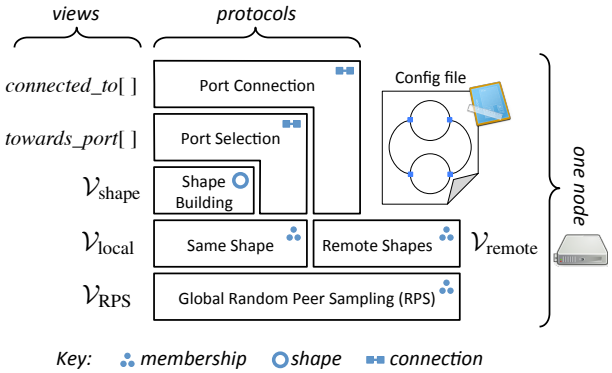


Figure 2: PLEIADES consists of 6 self-stabilizing protocols that build upon one another to enforce the structural invariant described in a configuration file distributed to all nodes in the system.

of these protocols also produces a self-stabilizing overlay. As such, each node maintains for each protocol a small set or array of other nodes in the system (called a *view*) that evolves in order to respect specific properties. The view maintained on a given node by each individual protocol is shown close to each rectangle (e.g. $\mathcal{V}_{\text{local}}$ for the *Same shape* protocol, and $towards\_port[\,]$ for the *Port selection* protocol). These protocols build on one another: higher protocols in Figure 2 use the view constructed by lower protocols to construct their own view.

In order to describe how these protocols collaborate to deliver PLEIADES, we must start by describing how individual shapes are described in our framework (in Section III-B), and how new nodes join individual shapes (Section III-C), before discussing first the membership and shape construction protocols (Section III-D), and finally turning to the Port Selection and Port Connection protocols (Section III-E).

B. Describing individual shapes

A shape s is a subset $N_{\mathsf{s}} \subseteq N$ of nodes organized in a particular *elementary topology*. Each shape follows a particular *template*, a reusable description of a shape's properties, that may be instantiated several times in a configuration file. (In Figure 2 for instance, the two rings of the configuration file would be two instances of the same template.) The structure enforced by a shape template tplate is captured by four pieces of information, that are used by the Shape Building protocol to realize the shape's elementary topology:

- the definition of a *position space* E_{tplate};
- a *projection function* $f_{\mathsf{tplate}} : N_{\mathsf{s}} \mapsto E_{\mathsf{tplate}}$ that assigns a position in E_{tplate} to each node selected to be part of an instance of tplate;
- a *ranking function*[3] $d_{\mathsf{tplate}} : E_{\mathsf{tplate}} \times E_{\mathsf{tplate}} \mapsto \mathbb{R}$;
- a *number of neighbors* (or shape fanout) per node, k_{tplate}.

[3]As mentioned in [17], self-organizing overlays employ ranking functions that cannot always be defined as global *distance* functions.

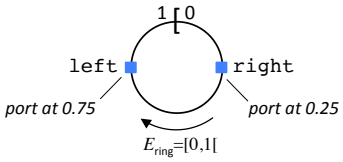


Figure 3: A simple ring template can be defined using $E_{\mathsf{ring}} = [0, 1[$ as position space, a random projection function, the modulo distance, and a shape fanout $k_{\mathsf{ring}} = 2$.

This information is sufficient for the Shape Building protocol to connect each node in $N_{\mathtt{s}}$ to its k_{tplate} closest neighbors according to the ranking function $d_{\mathsf{tplate}}()$.

For instance, a naive version of a self-stabilizing ring can be defined as follows (Figure 3):

$$\begin{aligned} E_{\mathsf{ring}} &= [0, 1[; \\ f_{\mathsf{ring}}(n) &= rand([0, 1[); \\ d_{\mathsf{ring}}(x, y) &= min(|x - y|, 1 - |x - y|); \\ k_{\mathsf{ring}} &= 2. \end{aligned}$$

This setting places nodes from N_{tplate} randomly on a circular identifier space, and selects the two closest instances of each node as its neighbors. (In practice, self-stabilizing rings typically seek to select $k_{\mathsf{tplate}}/2$ predecessors and $k_{\mathsf{tplate}}/2$ successors as neighbors of each node, to prevent clustering. See [17], [28], [31].)

In addition to its internal structure, a shape template also defines a set of *ports* to which other shapes may connect. In PLEIADES, a port is simply defined as a position in E_{tplate}, labeled with a name. Returning to the ring example of Figure 3, we may define two ports, named `left` and `right`, by associating them with the positions 0.25 and 0.75 within the identifier space $[0, 1[$.

C. Node joining procedure

When a node joins a running instance of PLEIADES, it decides which shape to contribute to based on the information of the configuration file. This joining mechanism may exploit a wide range of strategies, depending on uses cases: nodes in a particular location may be constrained to only join certain shapes, or nodes with certain properties may be forbidden to join certain shape templates. For simplicity, the version of PLEIADES we present in this paper uses a basic joining procedure, in which a new node randomly selects with equal probability one of the shapes of the configuration file. In Figure 2 for instance, this means half of the nodes on average would select the top ring, and the other half the bottom ring.

After it has selected its shape, a node populates the configuration variables shown in Table I, using the configuration file and the shape template definition. For instance, in the two-ring example of Figure 2, a node contributing to the top ring would pick a random id in $[0, 1[$, initialize the $shape.*$ variables from the ring template definition, and the $k.*$ variables regarding port connection from the configuration file. Here we would have for instance `left`.$remote\_shape.template$ = `ring` and `left`.$remote\_shape.remote\_port$ = `left`, meaning that the `left` port of the top ring should be connected to the `left` port of the bottom ring.

Table I: Configuration state of node n, in shape `s`

id	Position of node n in shape `s`;
$shape.id$	ID of the shape `s`;
$shape.template$	Template of the shape `s`;
$shape.ports$	List of shape `s`'s ports;
s	Maximum size of $\mathcal{V}_{\text{local}}$;
$\forall k \in shape.ports$:	
$k.id$	Position of port k in shape `s`;
$k.remote\_shape.id$	ID of remote shape linked to k;
$k.remote\_shape.template$	Template of remote shape linked to k;
$k.remote\_port$	Remote port linked to port k.

Table II: Views of membership and shape building prot.

$\mathcal{V}_{\text{RPS}}$	View of the Global RPS protocol;
$\mathcal{V}_{\text{local}}$	View of the Same Shape protocol `s`;
$\mathcal{V}_{\text{remote}}$	View of the Remote Shapes protocol;
$\mathcal{V}_{\text{shape}}$	View of shape `s`'s shape building protocol;

Table III: State of the connection protocols on node n

$\forall k \in shape.ports$:	
$is\_port[k]$	Boolean, whether n in charge of port k
$towards\_port[k]$	Local node that seems closest to port k
$connected\_to[k]$	Remote node that seems in charge of port k

D. The Membership and Shape Building protocols

Just after joining a shape, a node possesses no information about which other nodes belong to the same shape, or how to contact other nodes in other shapes. This information is provided by PLEIADES's three membership protocols. The Global *Random Peer-Sampling* (RPS) protocol [18] maintains, on each node, a continuously changing sample $\mathcal{V}_{\text{RPS}}$ of other nodes' *descriptors*. A node descriptor allows its complete identification on the system. It contains its network address, the ID of the shape it resides on, and its position on this shape.

This global peer sampling is then used to maintain two additional membership protocols: the *Same Shape Protocol* (*SSP*), and the *Remote Shapes Protocol* (*RSP*).

These two protocols, along with the list of neighbors returned by the Shape Building protocol, are used in turn by the *Port Selection* and *Port Connection* protocols (discussed in Section III-E), to create and maintain the links between the shapes according to the specification coded in the PLEIADES configuration file.

The notations of the views maintained by each node to implement the three membership protocols (Global RPS, Same Shape Protocol, and Remote Shapes Protocol) and the Shape Building protocol are summarized in Table II. We discuss each mechanism in turn in more detail in what follows. We take interest in a node n, that belongs to a shape `s`.

1) Global Random Peer Sampling (RPS): We assume that a RPS service is available for every node, and we simply emulate it in our experiments. Decentralized and efficient solutions exist, such as proposed by Jelasity *et al.* [18]. RPS protocols

Algorithm 1: *SSP*: Same Shape Protocol on n

Output: $n.\mathcal{V}_{\text{local}}$ converges to a s-sized sample of nodes from shape s

▷ Bootstrap by filtering the global peer sampling

1 $cand \leftarrow \{n' \in n.\mathcal{V}_{\text{RPS}} \mid n'.shape.id = n.shape.id\}$

2 $cand \leftarrow cand \cup n.\mathcal{V}_{\text{local}}$

▷ Exploit our neighbors' knowledge

3 **if** $cand \neq \varnothing$ **then**

4 $\quad q \leftarrow 1$ random node $\in cand$

$\quad$ ▷ Remote request to q

5 $\quad cand \leftarrow cand \cup q.\mathcal{V}_{\text{local}}$

6 **end**

▷ Truncation

7 $n.\mathcal{V}_{\text{local}} \leftarrow$ up to s random nodes $\in cand$

Algorithm 2: *RSP*: Remote Shapes Protocol on n

Output: $n.\mathcal{V}_{\text{remote}}$ converges to a view of one node per "close" shape.

▷ Bootstrap using the global peer sampling

1 $cand \leftarrow n.\mathcal{V}_{\text{remote}} \cup n.\mathcal{V}_{\text{RPS}}$

▷ Exploit other nodes' knowledge

2 **if** $cand \neq \varnothing$ **then**

3 $\quad q \leftarrow 1$ random node $\in cand$

$\quad$ ▷ Remote request to q

4 $\quad cand \leftarrow cand \cup q.\mathcal{V}_{\text{remote}}$

5 **end**

▷ Keep one node per "close" shape

6 **foreach** *close* shape $\mathsf{s}' \neq \mathsf{s}$ **do**

7 $\quad cand_{\mathsf{s}'} \leftarrow \{n' \in cand \mid n'.shape.id = \mathsf{s}'\}$

8 $\quad$ **if** $cand_{\mathsf{s}'} \neq \varnothing$ **then**

9 $\quad\quad n.\mathcal{V}_{\text{remote}}[\mathsf{s}'] \leftarrow 1$ random node $\in cand_{\mathsf{s}'}$

10 $\quad$ **end**

11 **end**

converge towards a constantly changing overlay that is close to a fixed-degree random graph. This graph shows a short diameter, which is useful to propagate or build distributed knowledge. This graph also remains connected with high probability, even under catastrophic failures, a particularly interesting property for our framework.

2) Same Shape Protocol (SSP): This overlay provides a node n with a view $\mathcal{V}_{\text{local}}$ of neighbors in the same shape s. The sub-procedure managing this overlay is shown in Algorithm 1. Upon bootstrap, $\mathcal{V}_{\text{local}}$ is empty. Each round, n takes candidate neighbors from the *Global RPS* overlay, keeping only nodes from its shape (line 1) in $cand$. It goes on merging its current $\mathcal{V}_{\text{local}}$ with the candidate set on line 2. If $cand$ is not empty (line 3), n selects a random neighbor q from $cand$ (line 4) and fetches q's local view, to add it to $cand$ (line 5). To limit memory consumption, the size of the local view $\mathcal{V}_{\text{local}}$ is bound to s elements (line 7).

If we assume the global peer-sampling overlay provides a uniformly distributed view of the complete system, we can calculate the average number of rounds to get at least s neighbors in function of the total number of nodes and shapes: the time to find the first neighbor is inversely proportional to the number of shapes, and the number of known neighbors then grows exponentially. In practice, simulations show that the size s needed for our framework is reached in a few rounds (Section IV) which allows the system to converge and reach a stable state quickly and efficiently.

3) Remote Shapes Protocol (RSP): This overlay is used to initiate inter-shape contacts. Upon bootstrap, $\mathcal{V}_{\text{remote}}$ is empty. During each round, the candidate set $cand$ is first filled with the previous content in the remote view $\mathcal{V}_{\text{remote}}$ and the global peer sampling view $\mathcal{V}_{\text{RPS}}$ on line 1. Then, n randomly picks a node q in $cand$ (line 3), fetches its remote view $q.\mathcal{V}_{\text{remote}}$, and adds it to its candidate set (line 4).

Lines 7 to 11 use the candidate set $cand$ to fill $n.\mathcal{V}_{\text{remote}}$ with one single descriptor per remote shape. To limit the memory consumption if the topology features many shapes, we propose to trim each node's remote view by keeping only descriptors from shapes that are considered *close* to s. This closeness metric is left to future work, but could be computed from the overall target topology or the shape's ID.

In detail, for each "close" shape s', line 7 filters candidate nodes from shape s' into $cand_{\mathsf{s}'}$, and lines 8-9 take a random node from $cand_{\mathsf{s}'}$ (if not empty) to fill $n.\mathcal{V}_{\text{remote}}[\mathsf{s}']$ (that is, the remote view's descriptor slot for shape s').

4) Shape Building Protocol: We use a variant of *Vicinity* [37] to organize the nodes that have joined a shape s into the basic topology prescribed by the shape's template tplate. (The pseudo-code is not shown for space reasons.) Vicinity uses a greedy push-pull procedure to populate each node n's view $\mathcal{V}_{\text{shape}}$ with close neighbors, according to the ranking function $d_{\mathsf{tplate}}()$, and then connects n to its k_{tplate} closest neighbors. Note that $\mathcal{V}_{\text{shape}}$'s size must be at least k_{tplate}, but in practice $\mathcal{V}_{\text{shape}}$ is usually larger, and we can bound its maximum size if we want to limit memory consumption. Vicinity exploits the transitivity of most ranking functions: if n is ranked close to o, and o is ranked close to p, then n is likely to be ranked close to p. However, whereas Vicinity uses a system-wide peer sampling protocol to find potential new neighbors, we restrict our Shape Building Protocol to the view $\mathcal{V}_{\text{local}}$ constructed by the *Same Shape Protocol*. This restriction to $\mathcal{V}_{\text{local}}$ insures the isolation and co-existence of multiple shapes in the same system.

E. The Port Selection and Connection protocols

The *Port Selection* procedure is executed between nodes within the same shape in order to determine which nodes are in charge of shape s's ports (these nodes are dubbed *port nodes*) while the *Port Connection* procedure is executed by port nodes to locate the remote port of the linked shape, and to establish

Algorithm 3: Port Selection on node n

Output: $is\_port[k]$ and $towards\_port[k]$ are greedily resolved for each port k in the shape s.

1 **foreach** $k \in n.shape.ports$ **do**
  ▷ Find closest node to port k among local nodes
2 $cand \leftarrow n.\mathcal{V}_{\text{local}} \cup n.\mathcal{V}_{\text{shape}} \cup \{n, n.towards\_port[k]\}$
3 $closest \leftarrow$ GETCLOSEST$(cand, k, n.shape.template)$
4 $n.is\_port[k] \leftarrow (n = closest)$
5 **if** $n.is\_port[k]$ **then**
6 $n.towards\_port[k] \leftarrow n$
7 **else**
  ▷ If n is not port node, remote request to *closest*
8 $n.towards\_port[k] \leftarrow closest.towards\_port[k]$
9 **end**
10 **end**

Function getClosest$(cand, k, \mathsf{tplate})$

Output: Returns the closest node from port k, among $cand$ nodes belonging to shapeof template tplate

1 $closest \leftarrow \arg\min_{p \in cand} (d_{\mathsf{tplate}}(p.id, k.id))$
2 **return** $closest$

Algorithm 4: Port Connection on node n

Output: n establishes a link with the node most likely in charge of k_2 within $dist\_shape$

1 **foreach** $k_1 \in n.shape.ports$ **do**
  ▷ Only executed by presumed port node for k_1
2 **if** $n.is\_port[k_1]$ **then**
3 $shape\_id \leftarrow k_1.remote\_shape.id$
4 $shape\_template \leftarrow k_1.remote\_shape.template$
5 $k_2 \leftarrow k_1.remote\_port$
  ▷ Closest remote node from k_2 that n knows of
6 $cand \leftarrow \{n.\mathcal{V}_{\text{remote}}[k_1], n.connected\_to[k_1]\}$
7 $closest \leftarrow$ GETCLOSEST$(cand, k_2, shape_template)$
  ▷ Remote request: who is the port node for k_2?
8 $n.connected\_to[k_1] \leftarrow closest.towards\_port[k_2]$
9 **end**
10 **end**

the link requested by the PLEIADES target specification. The variables used to maintain the state of the *Port Selection* and *Port Connection* protocols are shown in Table III.

1) GETCLOSEST$(cand, k, \mathsf{tplate})$*:* This function is used by both the Port Selection and Port Connection routines to find the closest node to a port. Given a set of nodes $cand$ and a port k, that all belong to the same shape s of template tplate, GETCLOSEST uses the shape template's rank function, d_{tplate} (see Section III-A), to measure the "distance" of each node in $cand$ to the port k. The function returns the node whose distance to port k is minimal.

2) Port Selection: We want each node n to know the port node of each of its shape's ports. The *Port Selection* routine maintains two variables for that purpose: for each port k of shape s, $towards\_port[k]$ contains the address of the presumed port node for k, and the $is\_port[k]$ flag is set when n believes it is in charge of k (in that case, $is\_port[k]$ points to n itself).

The variable $shape.ports$ contains the whole set of shape s's ports, given by the configuration. To fill $is\_port[k]$ and $towards\_port[k]$, n iterates over each port k in $shape.ports$ (line 1). By calling GETCLOSEST, n then checks which node is closest to the port k among all local nodes it knows of (lines 2-3). Candidates are taken from the local view $\mathcal{V}_{\text{local}}$ computed by *SSP*, from the Shape Building protocol's view $\mathcal{V}_{\text{shape}}$, in addition to n itself and the previous $towards\_port[k]$. n sets $is\_port[k]$ to true if it is the closest node to k, and to false otherwise (line 4). $towards\_port[k]$ is set to n if n seems to be the port node (line 6). Otherwise, n requests the *closest* node's own $towards\_port[k]$ (line 8), making $towards\_port[k]$ greedily converge to the port node for k.

3) Port Connection: When a node n believes it is in charge of a port k_1, it needs to find the other end of the topological link: the port node for k_2 in the remote shape (called s_2). The goal of the *Port Connection* routine, when n is in charge of a port k_1, is to maintain the $connected\_to[k_1]$ variable to the address of k_2's port node.

From lines 1 to 5, we iterate over each port k_1 in $shape.ports$, check that n is in charge of k_1, and create several variables: $shape\_id$ contains the ID of the linked shape s_2, $shape\_template$ is s_2's shape template, k_2 represents the remote port of k_1's link. n then picks the closest node to k_2 among two potential candidates (line 6): $\mathcal{V}_{\text{remote}}[k_1]$ (the random node from s_2 provided by *RSP*), and $connected\_to[k_1]$ (n's previous estimation of k_2's port node). It then calls the GETCLOSEST function on line 7, that will use the remote shape's ranking function to find the *closest* node to k_2 among the candidate set. Finally, on line 8, n requests *closest* for its $towards\_port[k_2]$ (leveraging the *Port Selection* procedure) to fill $n.connected\_to[k_1]$. This implementation again allows $connected\_to[k_1]$ to converge towards k_2's real port node in a greedy fashion.

IV. EVALUATION

In this section, we first discuss our evaluation set-up (Section IV-A) before briefly illustrating how PLEIADES can be used to create a range of advanced distributed structures (Section IV-B). We then evaluate the performance of

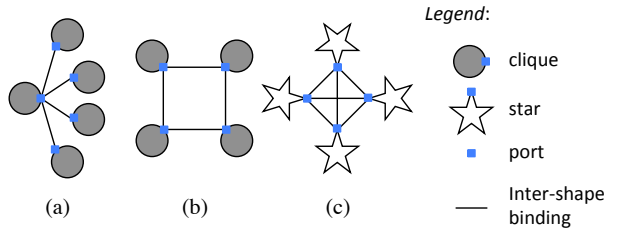


Figure 4: The PLEIADES configurations used to create the systems shown in Figure 5.

PLEIADES without reconfiguration or failures, in terms of convergence speed, scalability, and communication overhead (Section IV-C). Finally we test the reactions of PLEIADES under important perturbations, such as when a large portion of the system crashes, or an on-the-fly reconfiguration occurs (Section IV-D).

A. Evaluation set-up and methodology

We implemented the protocols that make up PLEIADES on top of PeerSim [27], except for the Global RPS protocol, which we emulated directly through PeerSim's API. We set the maximum size of $\mathcal{V}_{\text{local}}$ to 10, that of $\mathcal{V}_{\text{remote}}$ to the number of shapes in the systems, and we did not bound $\mathcal{V}_{\text{shape}}$, as in the original Vicinity protocol [37]. In order to demonstrate the capabilities of PLEIADES we created several shape templates (ring, star, clique) to serve as building blocks for more complex structural invariants. All experiments were averaged over 25 runs, to smooth the noise due to the probabilistic nature of gossip algorithms. We computed 90% confidence intervals but did not display them on the figures because they were too small to be readable.

B. Examples

Figure 4 graphically presents three configuration files used by PLEIADES to construct the three distributed systems shown in Figure 5. These three examples connect simpler shapes together (*cliques* and *stars*, shown symbolically in Figure 4 and with different colors in Figure 5). The resulting toplogies can be found in real-world applications, such as database sharding (Figure 5a), distributed key value stores (Figure 5b) or partially decentralized services using super-peers (Figure 5c).

These three examples illustrate PLEIADES's simplicity of use and expressiveness: a few basic shapes suffice to create an infinite number of variations that can be tailored to an application's needs.

C. Performances

PLEIADES targets very large systems using decentralized protocols. Decentralization, because it avoids any central point of coordination, and carries the risk of a degraded performance and/or high overhead. In the following we evaluate PLEIADES's performances in terms of *convergence speed* (Section IV-C1), *scalability* (Section IV-C2), and *communication overhead* (Section IV-C3).

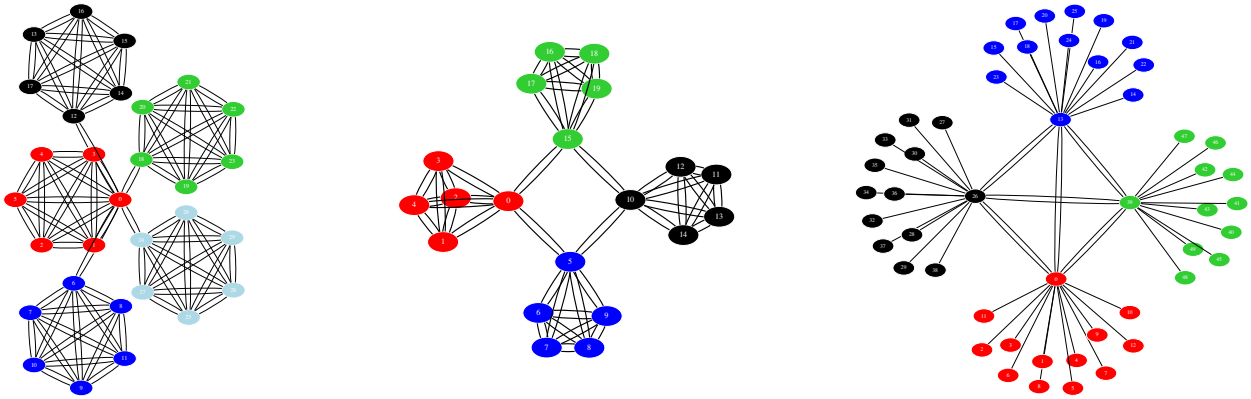
1) Convergence: We evaluate PLEIADES's convergence on a scenario comprising three rings connected into *a ring of rings*, whose configuration is represented in Figure 7. Figure 6 shows the execution of PLEIADES with this configuration on 100 nodes at three stages of the execution: after initialization (Fig. 6a), while the system is converging (Fig. 6b) and once converged (Fig. 6c). The overall system converges to the structure prescribed by its configuration in only 6 rounds. A round's duration is highly dependent on an application's needs, but setting for instance a round to 5s (a realistic assumption in light of PLEIADES's low communication costs as we will see in Section IV-C3), 6 rounds would correspond to a convergence time of 30s to organize 100 nodes from an arbitrary starting state. This time is comparable to the boot up time of a virtual machine on a public cloud.

Figure 8 shows the progress of the various sub-protocols that constitute PLEIADES on a ring of rings with a larger systems of 25,600 nodes, and a larger configuration comprising 10 rings. The figure charts over time the proportion of nodes in the correct state for a given protocol, from the point of view of a global omniscient observer. Except for the *Port Connection Protocol*, all protocols experience a rapid phase shift once they start converging, as is common in decentralized greedy protocols [17], [37]. The sequence of convergence roughly follows the dependencies between the protocols illustrated in Figure 2: the membership protocols *Remote Shapes* (RSP) and *Same Shape* (SSP) are the first to converge, followed by the *Shape Building* protocol (which depends on SSP), and the *Port Selection* protocol (which depends on Shape Building and on SSP).

The *Port Connection* protocol shows a less regular progression. The peak around round 4 is due to a few nodes that briefly believe they are ports (because *Port Selection* has not converged yet), and erroneously connect to remote shapes, thus falsely increasing our metric. In other words, *Port Connection* briefly converges to a local maximum but quickly escapes it when *Port Selection* starts to converge. Note however how ports get successfully connected even though the routing information provided by the *Port Selection* protocol is not fully converged yet: after 10 rounds, both the individual rings (*Shape Building*) and their connections (*Port Connection*) are in place to about 90%.

2) Scalability: PLEIADES scales well when the number of nodes and shapes in the system augments. We measured the convergence time of the system in rounds for a large variety of configurations, according to the following convergence criteria:

- *Same Shape Protocol* (SSP): at least 90% of the nodes have found 10 neighbours in the same shape;
- *Remote Shapes Protocol* (RSP): at least 90% of the nodes have found a node in each shape;
- *Shape Building Protocol*: at least 90% of the nodes have found their 2 closest neighbours in the ring;
- *Port Selection Protocol*: at least 90% of the ports are assigned to the correct node (and only this one);

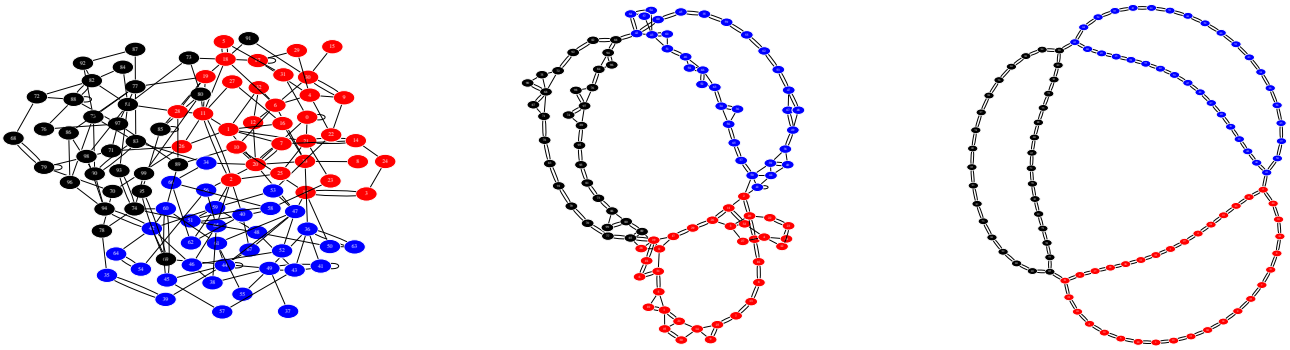


(a) A star of 5 Clique shapes, similar to topologies used in database sharding.

(b) A ring of 4 Clique shapes, similar to topologies used in distributed key-value stores.

(c) A clique of 4 Star shapes, similar to topologies used in partially decentralized services with super-peers.

Figure 5: The result topologies corresponding to the configurations of Figure 4 (after 10 rounds of simulation).



(a) Random initial state

(b) After 2 rounds, the general shape emerges.

(c) After 6 rounds, the system has converged.

Figure 6: A system of 100 nodes converges in 6 rounds towards three connected rings (colored in blue, red, and black).

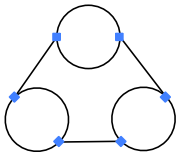


Figure 7: The PLEIADES configuration used in Figure 6.

- *Port Connection Protocol*: at least 90% of the ports found their related port in the remote shape.

In Figure 10, a configuration with 20 rings linked together sequentially is deployed for different number of nodes. All protocols converge in a few rounds, even for large number of nodes. Most importantly, they converge as fast or faster than the *Shape Building* protocol. Hence, the target complex topology is achieved sensibly at the same time as the local basic shapes.

It is interesting to note that the *Remote Shape* protocol (RSP) converges in constant time as the number of nodes augments. This is due to the fact that the ratio nodes/shapes is constant, so independently the total number of nodes in the system, it is as likely to find a node in a given shape. The abnormally high point for the *Shape Building* protocol (SSP) at 200 nodes is due to the fact that there are exactly 10 nodes per shape; so the convergence criterion used means that a node must have found all other nodes in the shape. But in practice, finding 6 or 7 of them is enough and does not hinder the convergence of the other protocols, as depicted on the graph. For larger numbers of nodes per shape, the convergence time is roughly constant, for the same reason as for RSP.

The other two protocols scale logarithmically with the number of nodes, similar to the *Shape Building* protocol.

In Figure 11, various configurations are deployed on a system of 25,600 nodes. Convergence time increases slowly with the number of shapes involved in the system, and even a complex system with 20 shapes converges in less than 15 rounds.

3) Communication overhead: Compared to an ad-hoc approach optimized for a given problem, PLEIADES incurs some

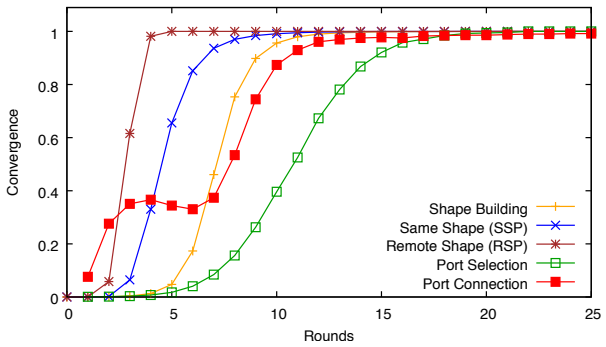


Figure 8: Progress of the different protocols of PLEIADES over time (in rounds) for a ring of rings with 25,600 nodes and 10 rings. Except for Port Connection, all protocols experience a rapid phase change.

overhead. This is the price to pay for a simpler and more systematic way to design topologies. In the following, we make the (very generous) assumption that an ad-hoc approach would not cost anything more than the resources needed to create the basic shapes, and we use the costs from the Shape Building protocol as our baseline.

For these measures, we considered that: *(i)* a node ID would use 16 bytes (IPV6 address); *(ii)* a node "position" would use 8 bytes (64-bit double); *(iii)* a shape ID would use 8 bytes (64-bit integer).

First, Figure 12 shows that the bandwidth consumption pattern over time is similar for the baseline and the overhead. Both rapidly reach a state where their bandwidth consumption per round and per node is stable. The actual values are also pretty low. For 25,600 nodes and 20 shapes, the bandwidth consumption per round is around 1,800 bytes, all combined.

The overhead is, of course, dependent on the complexity of the target topology. The more shapes and ports there are, the more messages are used to find and connect them. But even with large numbers of shapes, the overhead remains of a magnitude similar to the baseline. Figure 13 shows the ratio between baseline and overhead for different numbers of shapes on a system of 25,600 nodes in its stable state. This is measured once the system has converged because it is when nodes have discovered all their neighbors that the messages exchanged are heavier and the bandwidth consumption is the highest. It increases linearly with the number of shapes. As depicted in Figure 13 for 50 shapes, the bandwidth ratio is around 2, which in absolute value represents 1900 bytes, so it represents a very negligible amount.

D. Resilience

In the previous section, we showed that PLEIADES performs well under normal circumstances. In this section, we now consider how it reacts when heavily stressed. We used two scenarii: firstly, a dramatic crash where about half the nodes shut down (paragraph IV-D1); secondly, an on-the-fly reconfiguration of the target topology, changing the number of basic shapes in the system (paragraph IV-D2).

1) Dramatic crash: PLEIADES is extremely resilient, even in presence of catastrophic failures. To analyze this, a configuration with 4 shapes is deployed over different numbers of nodes, and stressed with various dramatic events, as illustrated in Figure 15.

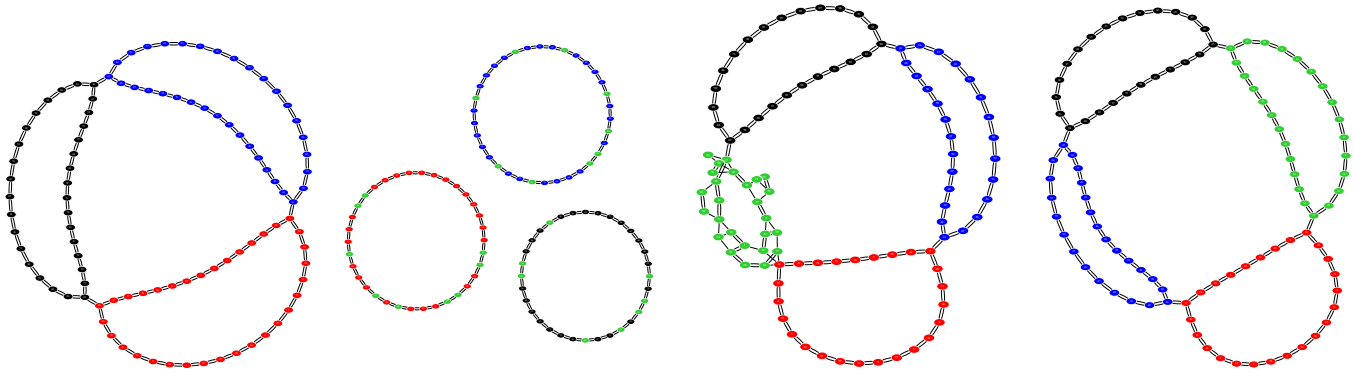
At first, we let the system converge as in the previous experiments. Then, we make each node crash with a probability $p = 0.5$, resulting in half the nodes crashing simultaneously on average and a totally broken topology (15a), and we let the system converge towards the new resulting target topology (15b). Finally, we simultaneously inject as many nodes as crashed earlier (15c) and we let the system converge back to the original target topology (15d). We consider two modes of reparation, either restoring crashed nodes to their last known state with a back-up, or providing new blank nodes initialized with random neighbors.

At each step, we measure the convergence time in rounds. For this experiment, we consider the system as a whole is converged when *all* the criteria in subsection IV-C2 are satisfied. Figure 14 plots the results: as shown previously, the initial convergence is quite fast and grows logarithmically with the number of nodes in the system: around 10 rounds even for very large systems of 20,000+ nodes.

More importantly, both the self-repair after crash and the return to the original target are faster than the initial convergence, even with such a dramatic rate of failure as we chose: they converge 2 to 5 rounds faster. Indeed, the nodes that are still online don't start with the same blank state as for the initial convergence, and this additional information more than compensates the stress caused by the crashes or re-injection, which enables the system to converge extremely fast.

2) Dynamic Reconfiguration: We argued that PLEIADES would help composing complex systems-of-systems and promote re-using previous works. But that means PLEIADES will need to be deployed to real systems that do not start in a random state.

We tried to *dynamically reconfigure* a system that was already deployed and converged to a stable state. For that, we need to define a *reconfiguration policy* that maps the relation between previous and current shape assignment. We shifted from a system with 3 shapes to 4 shapes, so we randomly assigned 1/3 of the nodes in each shape to the new shape. Many other policies may be envisioned, but due to space constraints we will only consider this one. At a given round (Figure 9b), the new configuration is sent to all the nodes, and some of them are allocated to the new shape. Only 2 rounds later (Figure 9c), the nodes in the new shape already found each others, and the previous shapes restored their stable state almost perfectly, despite losing some neighbors. A new stable state is rapidly reached (Figure 9d). All measurements presented in Section IV-C1 revealed that performances are at least as good for a dynamic reconfiguration from a converged state than for a system deployed from a random initial state. As with the crash scenario, this is due to some nodes—those not affected by the reconfiguration—starting with more information than with a random start.



(a) The system is deployed and converged to a stable state.

(b) When a new configuration is deployed, inter-shape links are reset, and nodes may be assigned to a new shape.

(c) After 2 rounds, nodes that did not change shape are already converged, and the newly introduced shape is starting to form.

(d) The system reaches its new stable state after 5 rounds, faster than from a random start.

Figure 9: Dynamic reconfiguration and convergence to a new stable state.

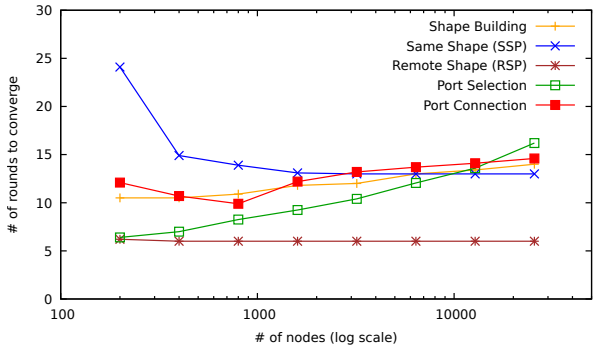


Figure 10: Convergence time of the PLEIADES protocols for a system of 20 connected rings (a *ring of rings*), for various system sizes. PLEIADES converges rapidly and scales well with the number of nodes.

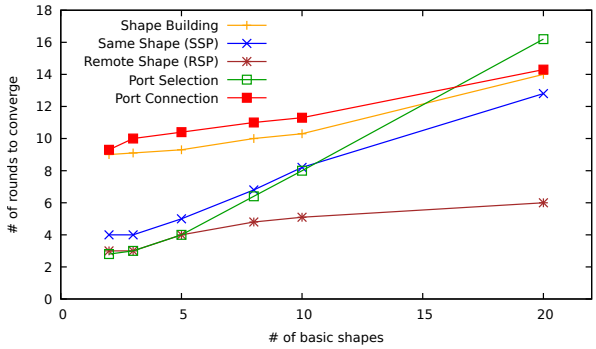


Figure 11: Convergence time of the PLEIADES protocols for a system of 25,600 nodes implementing a *ring of rings*, for various numbers of rings. The convergence time of PLEIADES only slowly increases with the number of individual rings.

To conclude, PLEIADES is extremely resilient, even in dramatic scenarii where a large proportion of the network is affected (up to 50%). The most difficult case is actually the initial cold start, because nodes start with very little information. In all other scenarii we tested, at least some nodes keep their knowledge of the network, which is enough to speed up the process.

V. RELATED WORK

Wireless Sensor Networks (WSN) have been a fertile ground for *holistic programming frameworks* aiming to simplify the programming of a very large number of distributed entities, as PLEIADES seeks to achieve.

Among them, approaches such as Kairos [14] and Regiment [30] draw their inspiration from existing distributed programming models. They provide means to quantify over multiple nodes, and hide the details of inter-node communication and coordination. Adopting a different stance, acquisitional query processors (e.g. TinyDB, Cougar, MauveDB) completely hide individual nodes, and provide a usually declarative approach to express which kind of data to sense, when, where and how often to sense and to aggregate it [11], [21], [4]. Sensing queries are then transparently mapped onto the WSN, taking into account various constraints such as energy consumption and reliability. Both node-dependent macro-programming approaches and acquisitional query processors move away from individual nodes and towards holistic programming abstractions. None of them however is able to maintain the distributed structural invariants supported by PLEIADES.

Originally proposed in the context of fixed networks [12], tuple-spaces provide a shared memory data abstraction to distributed systems in which tuples can be written to, read from, and queried by individual nodes. The model has been ported to

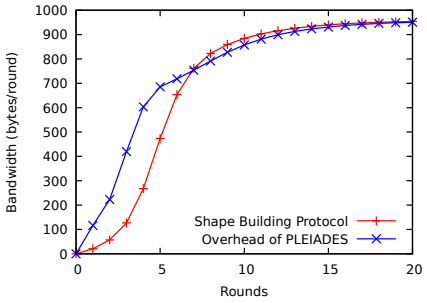


Figure 12: Bandwidth overhead of PLEIADES over the shape building protocol, per node, per round (20 shapes, 25,600 nodes). Both protocols peak once all views have stabilized, and remain below 1kB (2kB in total).

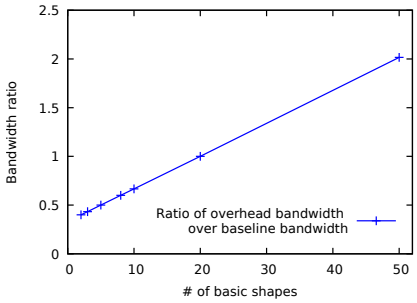


Figure 13: Evolution of the bandwidth overhead of PLEIADES (ratio) vs. the number of basic shapes (25,600 nodes, stable state). PLEIADES's overhead remains very small even for 50 basic shapes ($<$ 2kB in absolute value).

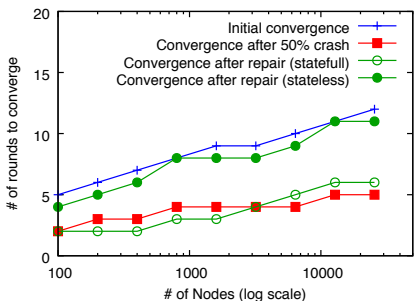
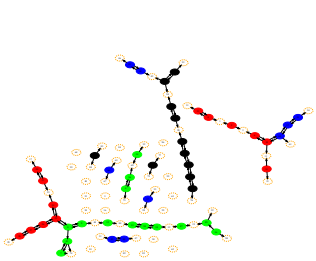
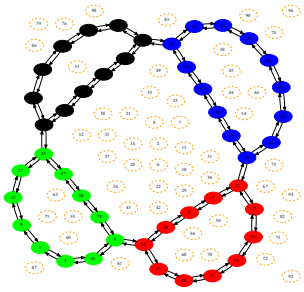


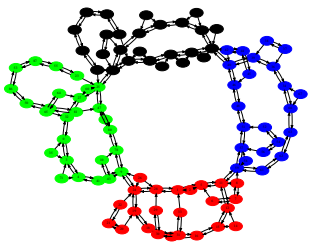
Figure 14: PLEIADES's convergence time after half of the nodes have crashed, and after re-injecting new nodes (4 connected rings, note the log x axis). PLEIADES's stabilization speed is logarithmic in the system's size.



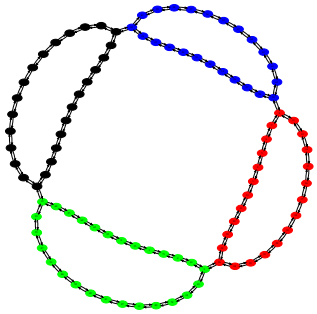
(a) Half the nodes crash (represented with a dashed line). The topology is completely broken.



(b) After 3 rounds, the system's structure has been reestablished.



(c) After reinjecting the crashed nodes, the original structural invariant has been lost.



(d) But after just 3 rouds the system is back to the original target, even faster than during the initial bootstrap phase.

Figure 15: Resilience and self-repair after a dramatic crash or a large node injection.

more dynamic systems with TineeLime [9], and TOTA (Tuple On The Air) [23]. Tuple-spaces are however more a high-level coordination model than a mean of organizing a large number of pre-existing entities as we do.

Neighborhood primitives such as Hood [40], Abstract Regions [39], and Logical Neighborhoods [29] are complementary to tuple-spaces. They provide scoping mechanisms that limit communication to sets of nodes (regions, or neighborhoods) selected according to a wide range of criteria. They are largely orthogonal to our approach, and could be exploited for instance to refine the shape joining mechanism of PLEIADES.

PLEIADES bears some similarity to Fragmented Objects [19], [22], in which a component's state is distributed (fragmented) among a number of distributed nodes in a manner that is fully transparent to its users. Fragmentation distributes a component's locus of computation, allowing for components to thus execute concurrently in a fully distributed manner. By relying on code mobility and state transfer mechanisms, fragmented objects can allow a component to extend or retract according to the current system's needs. However, implementations of fragmented components proposed so far [19] tend to be heavy-weight. They also typically rely solely on Remote Procedure Calls (RPC), an interaction paradigm that is ill-suited to loosely coupled large-scale systems.

PLEIADES can also be seen as a concrete example of some of the high-level capabilities sketched out by Blair *et al.* for *Holons* [3], a new paradigm for programming large-scale distributed systems relying on autonomous self-organization and opportunistic interactions.

VI. CONCLUSION & FUTURE WORKS

Large-scale distributed systems are becoming omnipresent while growing in size and complexity. Specifying and implementing such systems in a resilient manner is becoming increasingly tiresome and cumbersome for developers.

To address this challenge, we have introduced the PLEIADES framework. PLEIADES follows a programming-by-assembly design, while exploiting self-organizing overlays. However, PLEIADES goes a step further by considering *elementary shapes* as collective distributed entities and by enabling the creation of resilient, scalable, and complex distributed structural invariants through the assembly of these shapes.

To reach this aim, the PLEIADES framework combines six self-organizing protocols that work together to construct and maintain the structure prescribed in a PLEIADES configuration file. The resulting system is able to recover from catastrophic crash failures—such as the loss of a majority of the system's nodes—in only a few rounds while consuming a very limited bandwidth. PLEIADES further scales logarithmically in the number of system's nodes, and close to linearly in the number of elementary shapes.

We are currently designing a Domain Specific Language (DSL) to further increase the ease of programming of complex reliable large-scale distributed systems, and to strengthen the programming-by-assembly design of our approach. We are also integrating our approach on top of Kubernetes to augment Kubernetes's basic structural properties.

ACKNOWLEDGMENTS

This work was partially funded by the PAMELA (ANR-16-CE23-0016) and O'Browser (ANR-16-CE25-0005) projects of the French Agence Nationale de la Recherche (ANR), and by the DeSceNt project granted by the Labex CominLabs excellence laboratory of the French ANR (ANR-10-LABX-07-01). It has also received funding from CHIST-ERA under project DIONASYS, from the Swiss National Science Foundation (SNSF) and ANR.

REFERENCES

[1] R. Baraglia, P. Dazzi, M. Mordacchini, and L. Ricci. A peer-to-peer recommender system for self-emerging user communities based on gossip overlays. *J. of Comp. and System Sciences*, 79(2), 2013.

[2] M. Bertier, D. Frey, R. Guerraoui, A.-M. Kermarrec, and V. Leroy. The gossple anonymous social network. In *Middleware*, 2010.

[3] G. Blair, Y.-D. Bromberg, G. Coulson, Y. Elkhatib, L. Réveillère, H. B. Ribeiro, E. Rivière, and F. Taïani. Holons: Towards a systematic approach to composing systems of systems. In *Int. Workshop on Adaptive and Reflective Middleware*, ARM, 2015.

[4] P. Bonnet, J. Gehrke, and P. Seshadri. Towards sensor database systems. In *MDM '01: Second International Conference on Mobile Data Management*, pages 3–14, London, UK, 2001. Springer-Verlag.

[5] S. Bouget, H. Kervadec, A.-M. Kermarrec, and F. Taïani. Polystyrene: The decentralized data shape that never dies. In *2014 IEEE 34th ICDCS*, pages 288–297. IEEE, 2014.

[6] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The FRACTAL component model and its support in Java. *S:P&E*, 2006.

[7] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes. Borg, Omega, and Kubernetes. *Communications of the ACM*, 59(5), 2016.

[8] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Designing Privacy Enhancing Technologies*, pages 46–66, 2001.

[9] P. Costa, L. Mottola, A. L. Murphy, and G. P. Picco. Programming wireless sensor networks with the TeenyLime middleware. In *Middleware*, 2007.

[10] H. Deng and J. Xu. *CorePeer: A P2P Mechanism for Hybrid CDN-P2P Architecture*, pages 278–286. 2013.

[11] A. Deshpande and S. Madden. Mauvedb: supporting model-based user views in database systems. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. ACM, 2006.

[12] D. Gelernter. Generative communication in linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985.

[13] L. Glendenning, I. Beschastnikh, A. Krishnamurthy, and T. Anderson. Scalable consistency in Scatter. *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 15–28, 2011.

[14] R. Gummadi, O. Gnawali, and R. Govindan. Macro-programming wireless sensor networks using kairos. In *International Conference on Distributed Computing in Sensor Systems (DCOSS)*, 2005.

[15] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11. USENIX Association, 2011.

[16] Q. Huang, K. Birman, R. van Renesse, W. Lloyd, S. Kumar, and H. C. Li. An analysis of facebook photo caching. In *SOSP*, 2013.

[17] M. Jelasity, A. Montresor, and O. Babaoglu. T-Man: Gossip-based fast overlay topology construction. *Computer Networks*, 53(13), Aug. 2009.

[18] M. Jelasity, S. Voulgaris, R. Guerraoui, A.-M. Kermarrec, and M. Van Steen. Gossip-based peer sampling. *ACM TOCS*, 25(3):8, 2007.

[19] R. Kapitza, J. Domaschka, F. J. Hauck, H. P. Reiser, and H. Schmidt. Formi: Integrating adaptive fragmented objects into java rmi. *IEEE Distributed Systems Online*, 7(10), 2006.

[20] A.-M. Kermarrec, L. Massoulie, and A. Ganesh. Probabilistic reliable dissemination in large-scale systems. *IEEE TPDS*, 14(3), 2003.

[21] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tinydb: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1):122–173, 2005.

[22] M. Makpangou, Y. Gourhant, J.-P. Le Narzul, and M. Shapiro. Fragmented objects for distributed abstractions. In *Readings in Distributed Computing Systems*. July 1994.

[23] M. Mamei and F. Zambonelli. Programming pervasive and mobile computing applications: the tota approach. *ACM TSEM*, 2009.

[24] G. Mega, A. Montresor, and G. P. Picco. Efficient dissemination in decentralized social networks. In *P2P*, 2011.

[25] D. Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, 2014.

[26] MongoDB Inc. *MongoDB Manual (version 3.2) / Sharded Cluster Query Routing*. accessed 11 May 2016, https://docs.mongodb.com/manual/core/sharded-cluster-query-router/.

[27] A. Montresor and M. Jelasity. PeerSim: A scalable P2P simulator. In *P2P*, 2009.

[28] A. Montresor, M. Jelasity, and O. Babaoglu. Chord on demand. In *Proc. of the IEEE Int. Conf. on Peer-to-Peer Comp (P2P'05)*. IEEE, 2005.

[29] L. Mottola and G. P. Picco. Programming wireless sensor networks with logical neighborhoods. In *InterSense '06: Proceedings of the first international conference on Integrated internet ad hoc and sensor networks*, New York, NY, USA, 2006. ACM.

[30] R. Newton, G. Morrisett, and M. Welsh. The regiment macroprogramming system. In *IPSN '07: Proceedings of the 6th international conference on Information processing in sensor networks*, pages 489–498, New York, NY, USA, 2007. ACM.

[31] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup protocol for Internet applications. *IEEE/ACM Transactions on Networking*, 11(1):17–32, 2003.

[32] F. Taiani, S. Lin, and G. Blair. GossipKit: A Unified Component Framework for Gossip. *IEEE Trans. on Soft. Eng.*, 40(2), 2014.

[33] B. Technologies. *Riak KV Usage Reference / V3 Multi-Datacenter Replication Reference: Architecture*. accessed 11 May 2016, http://docs.basho.com/riak/kv/2.1.4/using/reference/v3-multi-datacenter/architecture/.

[34] J. Thones. Microservices. *Software, IEEE*, 32(1):116–116, 2015.

[35] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at Google with Borg. In *EuroSys*. ACM, 2015.

[36] S. Voulgaris and M. v. Steen. Epidemic-style management of semantic overlays for content-based searching. In *Euro-Par 2005 Parallel Processing*. Springer Berlin Heidelberg, 2005.

[37] S. Voulgaris and M. van Steen. Vicinity: A pinch of randomness brings out the structure. In *Middleware 2013*, pages 21–40. Springer, 2013.

[38] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn. Crush: Controlled, scalable, decentralized placement of replicated data. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[39] M. Welsh and G. Mainland. Programming sensor networks using abstract regions. In *First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI '04)*, pages 29–42, 2004.

[40] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler. Hood: a neighborhood abstraction for sensor networks. In *MobiSys*, 2004.

[41] H. Yin, X. Liu, T. Zhan, V. Sekar, F. Qiu, C. Lin, H. Zhang, and B. Li. LiveSky: Enhancing CDN with P2P. *ACM Trans. on Multimedia Comp. Comm. & App.*, 6:16:1–16:19, 2010.