Dolmen: Towards the programmatic assembly of large-scale distributed systems

Simon Bouget, Yérom-David Bromberg, François Taiani
University of Rennes 1 / IRISA, Rennes, France
{francois.taiani,simon.bouget,david.bromberg}@irisa.fr

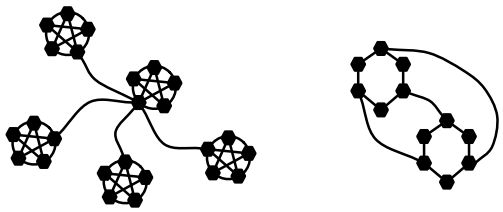
ABSTRACT

Large scale distributed systems have become ubiquitous, ranging from on-line social networks to the Internet-of-things, through storage and streaming platforms. To meet rising expectations in terms of scalability, robustness, and flexibility, these systems increasingly espouse complex distributed architectures, that are hard to design, deploy and maintain. To deal with this complexity, we argue that developers should be allowed to assemble programmatically large distributed systems from smaller parts using a seamless, high-level programming paradigm. To support this vision, we present Dolmen, a new assembly-based programming framework for the implementation of complex distributed topologies. Dolmen allows developers to easily define and realize complex distributed topologies as an assemblage of simpler blocks (e.g. rings, grids). It does so by harnessing the power of self-organizing overlays, that is made accessible to developers through a high-level Domain Specific Language and self-stabilizing runtime. Our evaluation further shows that Dolmen is generic, expressive, low-overhead and robust.

1. INTRODUCTION

Modern distributed applications are becoming increasing large and complex. They often bring together independently developed sub-systems (e.g. for storage, batch processing, streaming, application logic, logging, caching) into large, geo-distributed and heterogeneous architectures [15]. Combining, configuring, and deploying these architectures is a difficult and multifaceted task: individual services have their own requirements, configuration spaces, programming models, distribution logic, which must be carefully tuned to insure the overall performance, resilience, and evolvability of the resulting system.

This integration effort remains today largely an ad-hoc activity, that is either manual or uses tool-specific scripting capabilities. This low-level approach unfortunately scales poorly in the face of the increasingly complex deployment requirements and topologies of the involved services [20, 13, 27, 34]. For instance, *MongoDB* [27], a popular document-oriented no-sql databases, uses a star topology between sets of nodes organized in cliques (Figure 1a). Similarly the cross-datacenter replication feature of *Riak* [34], a production-level key-value datastore, relies on the connection of multiple rings across geo-distributed datacenters (Figure 1b). These services are often further embedded within micro-service architectures [35] resulting in increasingly complex distributed topologies, that can be hard to describe, monitor, and adapt.



(a) MongoDB (b) Riak

Figure 1: Some complex topologies encountered in modern distributed systems

This state of affairs imposes a high toll on developers. In order to write and maintain the low level glue code or configuration files required to realize these topologies, they must (i) have a *deep understanding* of the involved distributed services, their specific semantics, and individual programming model ; (ii) cater for the *unavoidable volatility* of the workloads and of the cloud infrastructures in which these services typically operate; and (iii) allow for a *continuous integration* process in which a deployed system is modified on the fly.

Easing the development of complex distributed systems has been a long-running and recurrent objective of middleware research. Most of these efforts have however focused on the local behavior of individual nodes (e.g. with protocol kernels [36, 26], or component frameworks [10, 6, 32]), rather than on the programmatic means to describe a system's global structure and behavior. As a result, most of these programming frameworks offer little or no support for the flexible integration of individual systems into a larger whole.

In order to fill this gap, we argue that practitioners should be allowed to programmatically manipulate distributed systems as *first class entities* [3], from which whole distributed systems can be *incrementally assembled*. We

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

argue that the mapping of systems to individual nodes should remain as much as possible transparent to developers. In particular developers should not have to worry about nodes failing, leaving or joining the system (a common occurrence in public clouds for instance), or about the intricacies of scaling operations.

As a first step towards this ambitious goal, we present in this paper DOLMEN, an assembly-based programming framework for the implementation of complex distributed topologies. DOLMEN provides developers with a high level component-based programming model [10, 6], and exploits self-organizing overlays [38, 2, 16] to map at runtime a developer's high-level description of a complex distributed topology onto a concrete infrastructure. DOLMEN relies on the scalability, resilience, and adaptability of self-organizing overlays to maintain a developer's target topology in the face of failures, scaling and dynamic adaptations.

DOLMEN goes beyond traditional component-based framework for distributed systems in that it considers components as *collective distributed entities* enforcing a given internal structure (a star, a tree, a ring) which developers can assemble programmatically to realize more complex topologies. It also goes beyond existing self-organizing overlays by supporting the description of a target topology as a *composition of more elementary shapes*, breaking away from the monolithic design of typical self-organizing overlay protocols.

Our contributions are as follows:

- We introduce a new programming model in which a community of distributed nodes organized in a particular topology can be manipulated as a first class entity, i.e a *components*, to incrementally construct more complex distributed structures ;
- We present DOLMEN, a component-based framework that implements our programming model. DOLMEN is capable of mapping a high-level representation of a target topology unto an actual infrastructure, while handling the intricacies involved in instantiating and composing distributed components within a dynamic environment.
- We implement a proof-of-concept of DOLMEN and perform a thorough evaluation that demonstrates its genericity, expressiveness, low-overhead, and adaptability.

The remainder of this paper is structured as follows. Section 2 presents the context and challenges motivating DOLMEN; Section 3 introduces our component-based programming model, and the framework built upon it, and explains how DOLMEN hides the intricacies of our approach from the programmer; Section 4 evaluates the scalability and efficiency of our framework with a proof-of-concept implementation; Section 5 discusses related work and Section 6 concludes.

2. PROBLEM, VISION, & BACKGROUND

2.1 Problem and vision

An 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 that each execute a Paxos instance to provide 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 set*, 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 [27]. *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 use special nodes, known as *fullsync coordinators* to handle the replication to each sink [34]. Application level system are experiencing a similar evolution, and are moving towards flexible, composite deployment topologies, a trend fueled the rapid rise of container-based micro-service architectures [35, 25].

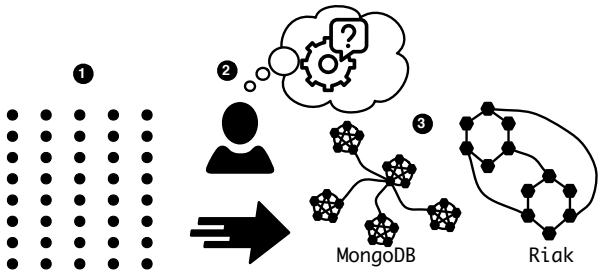


Figure 2: Creation of complex systems of systems

Provisioning, deploying and maintaining these distributed topologies is unfortunately a cumbersome and error-prone task. Developers must provision nodes (Figure 2 ❶), specify the local services they should execute (Figure 2 ❷), and connect these local services into an appropriate topology (Figure 2 ❸), according to the service and application's needs. Many of these tasks can be alleviated using deployment automation tools such as *Borg* [37], *Kubernetes* [7], *Aurora* or *Mesos*. These tools offer essential self-healing and scaling capabilities, but still require developers to manually configure the system's actual topology, potentially aided by low-level scripting tools. This ad-hoc approach results in makeshift, tedious and error-prone code, as this code must take into account the specificity of individual services, must account for the volatility of the cloud infrastructures in which these services typically operate, and must handle service-specific roll-back and recovery operations in case of failures or partitioning.

In this paper, we take a somewhat extreme stance, and argue that complex topology maintenance and construction should follow a *generic*, *principled* and *systematic* strategy. More precisely, we advocate a high-level declarative paradigm in which complex topologies can be manipulated as first class programmatic entities, and composed to form larger systems, abstracting away the individual nodes that compose them. As a first step in this direction we propose DOLMEN, an assembly-based topology programming framework that free developers from low-level topology deployment and maintenance. DOLMEN brings together two core ideas: *component-based programming*, a long running strategy for modular distributed development, and *self-organizing overlays*, an extenstion of the autonomous

and self-healing mechanisms found in some of today's production-grade environments. We discuss both of these core idea in turn in the following.

2.2 Component-based programming

Component-based software engineering (CBSE) promotes *development by assembly*. It allows developers to construct complex systems by assembling pre-existing *components*, i.e. modular reusable blocks that explicitly exposes their interfaces—both in terms of requirements and of features provided. Components provide *separation of concerns* and *modularity*, and facilitate re-use and continuous integration. A large number of component technologies have been successfully applied to distributed systems over the years, both in industry (e.g. *Enterprise Java Beans* (EJB), the *Service Component Architecture* (SCA), the *CORBA Component Model* (CCM), .Net, and the *OSGi Remote Services* Specification) and academia [6, 10].

These solutions, however, view components as software artifacts living *within* nodes, and focus therefore on the workings of individual nodes rather than on a system's global behavior. By contrast, we propose to inverse this view, and consider components as *distributed entities* enforcing a given internal structure (a star, a tree, a ring) which developers can assemble programmatically to realize more complex topologies. Individual nodes now live within components, and become transparent to developers, who only perceive system-level entities they can instantiate and connect to form larger wholes.

2.3 Self-organizing overlays

To realize this vision, we propose to exploit self-organizing overlays [16, 38], 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 [16] or Vicinity [38] 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 ring or a star of cliques, are more problematic. This model does not lend itself naturally to development by assembly: self-organizing overlays, in their basic form, have no notion of composition, bindings, or port.

2.4 Key challenges and roadmap

In the remaining of this paper, we present DOLMEN, an assembly-based topology programming framework that harnesses the autonomous properties of self-organizing overlays to realize a system-level component-based programming framework. To deliver this model, our solution must resolve a number of key challenges: (i) provide a high-level description of the target composite topology; (ii) map "system-level" components to nodes, (iii) realize the dynamic bindings that connect individual shapes according to the developer's high-level plan, (iv) maintain and handle communications among different components.

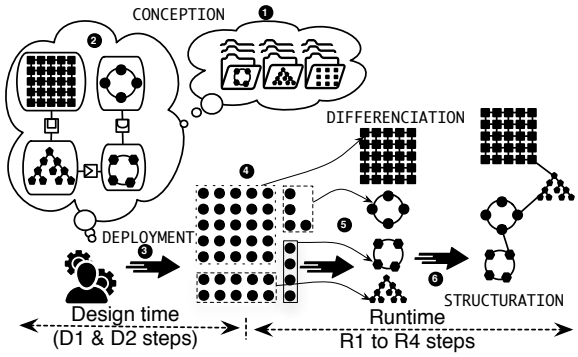


Figure 3: Dolmen overall approach

3. THE DOLMEN FRAMEWORK: DSL, LIBRARY & RUNTIME

The DOLMEN framework comprises (i) a DSL, (ii) a component library, and (iii) a runtime. The DOLMEN DSL is simple and expressive enough to describe a large array of topologies that can be difficult to achieve with earlier methods. The DOLMEN DSL achieves this goal by allowing developers to construct a complex topology by assembling simpler blocks, termed *components*. To support this process, the DOLMEN framework provides a component library that includes, by default, base components that implement basic topology shapes such as rings, grids, etc. Finally, the DOLMEN framework comes with a runtime that handles under-the-hood the role allocation and the differentiation of nodes that belong to different components.

In the following, we describe in detail the DOLMEN framework. Section 3.1 introduces the component library, and its underlying key notions such as node, overlay, component, etc. Section 3.2 presents the DSL, and illustrates its use to create complex overlays. Finally, Section 3.3 discusses the runtime implementation.

3.1 Component library

In DOLMEN, a component is a subset $N_c \subseteq \{n_i, i \in [1..n]\}$ of message-passing *nodes* organized in a particular *elementary topology*. The DOLMEN component library contains a predefined set of components implementing a range of such elementary topologies (a ring, a tree, a torus), that a developer can combine to build a complex distributed topology. This combination relies on *ports*, i.e. logical points of contact between different components, that are managed at runtime by (at least) one node in each involved component. Ports from two different components are connected through *links*. A link between two ports is a logical connection between two components, and at the node level, it is a link between two nodes assigned to two different components.

From an implementation point of view, components and links are implemented using multiple *layers* of overlays that are built upon each other: one self organizing overlay per component (known as the component's *core protocol*) realizes the component's actual shape, while two other overlays are used to locate ports, and realize links. The

system's *resulting overall topology* is the union of these different overlays.

The definition of a new component type C must provide the three pieces of information required by its core protocol to realize the component's target shape: an integer k_C, the definition of a *position space* E_C, a *distance*[1] function $dist_C : E_C \times E_C \mapsto \mathbb{R}$, and a projection function $f_C : N_C \mapsto E_C$ that assigns a position in E_C to each node selected to be part of C. The corresponding self-stabilizing overlay (in our case a variant of Vicinity [38]) uses the procedure described in Section 2.3 to connect the nodes participating in C (noted N_C) to the k_C closest other nodes in N_C according to $dist_C$.

For instance, a self-stabilizing ring can be realized by choosing

$$E_{\text{ring}} = [0, 1[,$$
$$f_{\text{ring}}(n) = rand([0, 1[), and$$
$$dist_{\text{ring}}(x, y) = min(|x - y|, 1 - |x - y|),$$

i.e. by placing nodes randomly in a circular identifier space, and selecting the k_{ring} closest node as neighbors for each node along the resulting ring. (In practice a topology in which each node is on average connected to $k_{\text{ring}}/2$ successors and $k_{\text{ring}}/2$ processors, as often encountered in ring-based DHTs [29, 33].)

In addition to its internal topology, each component needs to define a set of *ports* to which other components may connect. In DOLMEN, a port is simply defined as a position in E which is named. Returning to our ring example, we may define two ports, named `left` and `right`, by associating them with the positions 0.25 and 0.75 within the identifier space $[0, 1[$.

To summarize, to use the component library, developers must do the following at design time:

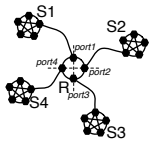
- **(D1)** use the DOLMEN DSL to describe which components should be instantiated, how they should be connected to each other, and provide node-provisioning policies to determine how individual node should be allocated to components (See Figure 3 ❶ ❷);
- **(D2)** compile and deploy the resulting DOLMEN configuration file to a set of nodes executing the DOLMEN runtime (Figure 3 ❸).

On receiving this file the runtime will

- **(R1)** allocate individual node to component instance (determining N_C for each component C): by default each node belongs to one and only one component instance ; (Figure 3 ❹)
- **(R2)** create each component's internal topology using its core protocol instance; and
- **(R3)** identify within each component which nodes should manage this component's individual ports ; (Figure 3 ❺)
- **(R4)** finally, connect the resulting ports according to the programmer's specification. (Figure 3 ❻)

These runtime steps occur in a fully decentralized manner, without resorting to any centralized entities, a key property

[1]Although it often does, this distance function might not in some cases fulfill all the usual properties of a *distance function* in a mathematical sense. We keep the name for ease of exposition.




```
1  import { Clique, Ring } from defaultShapes
2
3  component Router as r using Ring
4  r.nodes(random,0,2)
5  r.addPort(port1, 0)
6  r.addPort(port2, 0,25)
7  r.addPort(port3, 0,50)
8  r.addPort(port4, 0,75)
9
10 component Shard1 as s1 using Clique
11 s1.nodes(random,0,2)
12 s1.addPort(leader,0)
13
14 component Shard2 as s2 using Clique
15 s2.nodes(random,0,2)
16 s2.addPort(leader,0)
17
18 component Shard3 as s3 using Clique
19 s3.nodes(random,0,2)
20 s3.addPort(leader,0)
21
22 component Shard4 as s4 using Clique
23 s4.nodes(random,0,2)
24 s4.addPort(leader,0)
25
26
27 links {
28    (r.port1, s1.leader),
29    (r.port2, s2.leader),
30    (r.port3, s3.leader),
31    (r.port4, s4.leader)
32 }
```

Figure 4: MongoDB like topology: ring of cliques

regarding the scalability and resilience of our approach. In the following subsection, we present in more details the Dolmen DSL and its usage (Step **P1**), before describing how the runtime fulfills its missions (Steps **R1-4**) in Section 3.3.

3.2 DOLMEN DSL

The DOLMEN DSL enables programmers to describe their expected complex distributed topology, i.e the *target topology*, by specifying the components they want to link through *ports*. For instance, the code given in Figure 4 describes how to create a ring of cliques. The described topology is nearly similar to the ones actually used in current applications such as MongoDB to create shards. The difference comes from the fact that we use a ring instead of a star, solely to demonstrate how easy it is to assemble different kind of elementary topologies (e.g. one ring with cliques).

To create the target topology, i.e. a ring of cliques, elementary topologies need to be imported. So both a ring and a clique shapes are imported (line 1) from the component library provided by the DOLMEN framework. Next, two different type of components are created. In one hand, a component that: (i) acts as a router, and (ii) implements a ring topology (line 3). In the other hand, four others components that: (i) act as sharded clusters, and (ii) implement a clique topology (line 10,14,18,22). The keywords `component` and `using` enable to create a new component based on an existing elementary topology that has been previously imported. As a component is a subset of nodes, the next required step is to specify how the available nodes of our distributed system will be mapped at runtime to each defined component. For instance in a cloud system, where nodes represent virtual machines in servers, and are all functionally identical, the allocation can

be left to random. Accordingly, the `nodes(strategie,...)` method enables to specify the nodes mapping strategy for a component. It takes the strategy and its parameters as inputs. Currently, four different strategies are available: *random*, *bandwith*, *cpu* and *storage*. They enable to control how to allocate nodes to a component according to different *criteria* (i.e. characteristics of a node) with a given probability. So, for instance, to perform, at runtime, an uniform distribution of the available nodes among the five components, a random strategy with a percentage of 20% is set for each component (line 4, 11, 15, 19, 23). To further link components altogether, ports need to be created for each component. As the router component has a ring topology, a port is created, for instance, at the terminal side of each quadrantal angles of the ring, i.e. at the positions 0, 0.25, 0.5, and 0.75 within the identifier space $[0, 1[$ (See line 5-8). For the four other components, a port is setup at a random position in the position space of each component (line 11,15,19,23). Adding a port to a component at a specific position according to its position space is performed *via* the `addPort(label, position)` method. Once created, the port can be directly addressed by its *label* from the concerned component. Finally, to build the target topology, a list of links/connections between the aforementioned ports are created *via* the `links` keyword (line 27-32).

3.3 Runtime implementation

At runtime, the DOLMEN specification written in the DOLMEN DSL is disseminated to all participating nodes, where it is processed by the node's DOLMEN runtime. For brevity's sake, we do not discuss how this dissemination occurs: for instance, this step could rely on a gossip broadcast [19], or, in a cloud infrastructure, each node could retrieve the configuration from its original VM image. Because DOLMEN is self-stabilizing, nodes may receive this configuration at different points in time without impacting the system's eventual convergence.

The overall organization of the DOLMEN per-node runtime is shown in Figure 5. The runtime maintains a *global peer-sampling service* [17] that allows each node to obtain a continuously changing sample of other nodes participating in the system. This global peer sampling is then used to maintain two *Utility Overlays* (or *UO*): a *Same Component Overlay* (**UO1**), and a *Distant Component Overlay* (**UO2**). These two utility overlays, along with the list of neighbors returned by the Core Protocol (discussed in Section 3.1), are used in turn in two procedures, *port connection* and *port selection*, that create and maintain the links between the components according to the specification coded in the DOLMEN DSL.

As often encountered in self-organizing overlays, the two utility overlays and the two link procedures use a greedy optimization strategy organized in asynchronous rounds to progressively converge towards a desirable outcome. On every node, both overlays maintain in particular a fixed-size set of other nodes, known as a *view*, which captures the current node's neighbors in this overlay. Both overlays then regularly modify this view by interacting with neighboring nodes in order to fulfill the following roles:

- The *same-component overlay* is used to find other nodes in the same component. It provides in effect a peer-sampling service limited to the local component, i.e. a continuously changing view of nodes belonging to that component.
- The *distant component overlay* is used to find nodes in other components. It provides a continuously changing sample of nodes, belonging each to a different remote component.
- The *port selection* procedure is executed between nodes within the same component in order to determine which nodes are in charge of this component's ports.
- The *port connection* procedure is executed by nodes within a component to (i) locate *port nodes*, and (ii) connect this *port nodes* to those of other components according to the DOLMEN DSL specification.

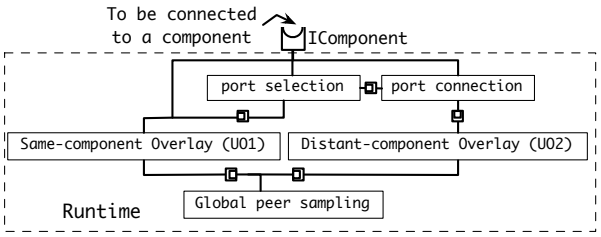


Figure 5: Organization of the Dolmen runtime

Table 1: A node local state

node_id	unique identifier of a node
comp_id	ID of the component this node belongs to
comp_id.type	type of the component this node belongs to
global_view	view of the Global peer sampling Overlay
same_view	view of the Same Component Overlay (UO1)
other_view	view of the Distant Component Overlay (UO2)
core_view	view of the Core Protocol overlay
port?	is in charge of a port or not
toward_port	nodes to contact to reach a port
connected_to	nodes in distant components a port is connected to

The variables maintained by each node to implement these four mechanisms (the two UOs and the two procedures) are summarized in Table 1. We discuss each mechanism in turn in more detail in what follows.

3.3.1 Same component overlay (UO1)

This overlay provides a node n with a list *same_view* of neighbors in the same component instance, i.e. $\forall p \in n.same\_view$, $p.comp\_id = n.comp\_id$. The sub-procedure managing this overlay is shown in Algorithm 1. Initially, *same_view* is empty, and for each round, k candidate neighbors are taken from the *Global peer sampling* overlay (line 1), and kept if they are indeed in the same component (line 2). If it has at least one neighbor (line 3), every node selects a random neighbor in the same component to retrieve its view. The view of the selected neighbor is added to the candidates (*cand* variable, line 5). To limit memory consumption, if more than s candidates are available, the local node randomly chooses s neighbors among them (line 8, truncation).

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 components. In practice, simulations show that the

Algorithm 1: UO1: Same Component overlay on n

Data: *same_view* progressively converges to a sample of nodes in the same component using a biased RPS procedure

▷ Bootstrapping using the global peer sampling

1 $cand \leftarrow n.same\_view \cup \{k \text{ rand. nodes} \in global\_view\}$
2 $cand \leftarrow \{n' \in cand \mid n'.comp\_id = n.comp\_id\}$

▷ Exploiting neighbors if we have some

3 **if** $n.same\_view \neq \emptyset$ **then**
4 $\quad q \leftarrow 1$ random node $\in n.same\_view$
5 $\quad cand \leftarrow cand \cup q.same\_view$
6 **end**

▷ Truncation

7 **if** $size(cand) > s$ **then**
8 $\quad cand \leftarrow s$ random nodes $\in cand$
9 **end**
10 $n.same\_view \leftarrow cand$

Algorithm 2: OU2: Distant Component Overlay on n

Data: *other_view* converges to a state in which it contains nodes of "neighboring" components, i.e. components with close *comp_id* values.

▷ Selecting candidate nodes

1 $q \leftarrow 1$ random node $\in global\_view$
2 $cand \leftarrow n.other\_view + q + q.other\_view$

▷ Filtering

3 Only keep one node per *comp_id* in *cand*
4 $cand \leftarrow \{n' \in cand$ with "close" $comp\_id\}$ ▷ optional
5 $other\_view = cand$

Algorithm 3: Port selection (on node n)

Data: $port?(k)$ and $toward\_port(k)$ for port k are progressively resolved using a greedy procedure.

1 **foreach** $k \in ports$ **do**
2 $\quad closest \leftarrow$ `best_for(`k`)`
3 $\quad$ **if** $n = closest$ **then**
4 $\quad\quad n.port?(k) \leftarrow true$
5 $\quad\quad n.toward\_port(k) \leftarrow n$
6 $\quad$ **else**
7 $\quad\quad n.port?(k) \leftarrow false$
8 $\quad\quad n.toward\_port(k) \leftarrow closest.toward\_port(k)$
9 $\quad$ **end**
10 **end**

11 **Procedure** `best_for(`*port: k*`)`
12 $\quad S \leftarrow n \cup n.core\_view \cup n.tward\_port(k) \cup n.same\_view$
13 $\quad \mathsf{ctype} \leftarrow n.comp\_id.type$
$\quad closest \leftarrow \mathsf{argmin}_{p \in S} \left(d_{\mathsf{ctype}}\left(f_{\mathsf{ctype}}\left(p\right), k\right)\right)$
14 $\quad$ **return** *closest*
15 **end**

Algorithm 4: Port connection (on node n)

Data: n establishes a link with the node most likely in charge of k_2 within *dist_comp*

1 **foreach** $k_1 \in ports$ **do**
2 $\quad$ **if** $n.port?(k_1) = true$ **then**
$\quad\quad$ ▷ *dist_comp* and k_2 are found in the DSL specification
3 $\quad\quad dist\_comp \leftarrow$ component that k_1 is connected to
4 $\quad\quad k2 \leftarrow$ port of *dist_comp* that k_1 connects to
5 $\quad\quad candidate \leftarrow$ node from *other_view* in *dist_comp*
6 $\quad\quad connected\_to(k_1) \leftarrow candidate.toward\_port(k_2)$
7 $\quad$ **end**
8 **end**

size s needed for our framework is reached relatively fast (Section 4) which allows the system to converge and reach a stable state efficiently.

3.3.2 Distant Component Overlay (UO2)

This overlay is used to initiate inter-component contacts. Initially, *other_view* is empty, and during each round one candidate neighbor is retrieved from the global peer-sampling overlay (line 1). Every node then exchanges its view with this random neighbor, and the neighbor's view is added to the candidates, as well as the current view (line 2). Finally, the view is updated by keeping one candidate for each component among the candidates (line 3-5), i.e. $\forall (p, q) \in n.other\_view, p.comp\_id \neq q.comp\_id$. To limit the memory consumption if the number of components is too large, the view can also be trimmed to keep only nodes in "close" components.

3.3.3 Port selection

For each port k of the component it is part of, each node must be able to decide whether it is in charge of this port. If not, the local node should route inter-component queries to the correct node. This selection and routing procedure relies on the function `best_for(`k`)` (lines 11-14) which finds the node closest to the position of port k in the component's position space E_{ctype} (where ctype denotes the type of the local component, as discussed in Section 3.1). `best_for(`k`)` considers the nodes known to n among its local core protocol view (*core_view*), its view in UO1 (*same_view*), and its current estimation of the node handling port k ($n.toward\_port(k)$), and returns the node p whose position $f_{\mathsf{ctype}}(p)$ lies closest to the position of port k in the current component.

If `best_for(`k`)` returns n (line 3), then n is in charge of port k (lines 4-5), else n contacts the node returned by `best_for(`k`)` to retrieve a node more likely in charge of port k. This greedy procedure converges rapidly so that for all nodes n and for all ports k, $n.toward\_port(k)$ returns the node in charge of k.

3.3.4 Port connection

When the configuration indicates a link between the port k_1 of the local component and the port k_2 of the component *dist_comp*, the node in charge of k_1 needs to find and contact the node in charge of k_2 in component *dist_comp*. This is achieved thanks to the sub-procedure *Port connection* shown in Algorithm 4.

If n is in charge of a port k_1, n uses *other_view* (i.e. the view from the distant component overlay) to contact a node belonging to the distant component *dist_comp* that

```
1  import { Clique } from defaultShapes
2
3  component Router as r using Clique
4  r.nodes(random,0,2)
5  r.addPort(leader, 0)
6
7  component Shard1 as s1 using Clique
8  s1.nodes(random,0,2)
9  s1.addPort(leader,0)
10
11 component Shard2 as s2 using Clique
12 s2.nodes(random,0,2)
13 s2.addPort(leader,0)
14
15 component Shard3 as s3 using Clique
16 s3.nodes(random,0,2)
17 s3.addPort(leader,0)
18
19 component Shard4 as s4 using Clique
20 s4.nodes(random,0,2)
21 s4.addPort(leader,0)
22
23
24 links {
25    (r.leader, s1.leader),
26    (r.leader, s2.leader),
27    (r.leader, s3.leader),
28    (r.leader, s4.leader)
29 }
```

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

```
1  import { Clique } from defaultShapes
2
3  component Cluster1 as c1 using Clique
4  c1.nodes(random,0,25)
5  c1.addPort(leader,0)
6
7  component Cluster2 as c2 using Clique
8  c2.nodes(random,0,25)
9  c2.addPort(leader,0)
10
11 component Cluster3 as c3 using Clique
12 c3.nodes(random,0,25)
13 c3.addPort(leader,0)
14
15 component Cluster4 as c4 using Clique
16 c4.nodes(random,0,25)
17 c4.addPort(leader,0)
18
19
20 links {
21   (c1.leader, c2.leader),
22   (c2.leader, c3.leader),
23   (c3.leader, c4.leader),
24   (c4.leader, c1.leader)
25 }
```

(b) A ring of 4 Cliques components, similar to topologies used in distributed storage

```
1  import { Star } from defaultShapes
2
3  component Cluster1 as c1 using Star
4  c1.nodes(random,0,25)
5  c1.addPort(leader,0)
6
7  component Cluster2 as c2 using Star
8  c2.nodes(random,0,25)
9  c2.addPort(leader,0)
10
11 component Cluster3 as c3 using Star
12 c3.nodes(random,0,25)
13 c3.addPort(leader,0)
14
15 component Cluster4 as c4 using Star
16 c4.nodes(random,0,25)
17 c4.addPort(leader,0)
18
19
20 links {
21   (c1.leader, c2.leader),
22   (c1.leader, c3.leader),
23   (c1.leader, c4.leader),
24   (c2.leader, c3.leader),
25   (c2.leader, c4.leader),
26   (c3.leader, c4.leader)
27 }
```

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

Figure 6: Describing target topologies is very easy. Completely different topologies are obtained just by changing a few lines.

provides Port k_2 (line 5). Then, as in Algorithm 3, it uses the routing pointer $toward\_port(k_2)$ to greedily converge toward the node actually in charge of distant port k_2 (line 6)—possibly over several rounds, although simulations show that it converges rapidly (see next section).

4. EXPERIMENTATIONS

Our goal in this section is to show the applicability of DOLMEN. We realized a proof-of-concept implementation of the runtime described in section 3.3. We also used the overlay-building algorithm Vicinity [38] to create a few basic shape components (Ring, Star, Clique) for the library described in 3.1. We then used them to show that our approach: (a) can actually generate complex topologies, comparable to those used currently in real-world applications; (b) is easy to use; (c) is efficient.

All experiments were run in the PeerSim simulator [28] and measures were averaged over 25 runs, to smooth the noise due to the probabilistic nature of gossip algorithms. We computed 90% confidence intervals but do not display them on the figures because they are too small to be readable.

In the rest of this section, we briefly show how DOLMEN can be used and the kind of topologies it can easily generate (section 4.1), then we evaluate the performance of the runtime regarding a set of different criteria such as convergence, dynamicity, scalability, and overhead (section 4.2).

4.1 Examples

This subsection aims to show that our framework can be used to create a vast array of topologies similar to those used in real-world applications, and that it is actually easy to do so.

We used DOLMEN to generate an array of different topologies, as shown in Figure 7. These topologies are representative of real-world applications, such as database sharding (7a), distributed storage systems (7b) or partially decentralized services using super-peers (7c).

Once the runtime and the component library are available, describing these different topologies with DOLMEN DSL is just a matter of a few lines, as shown in Figure 6.

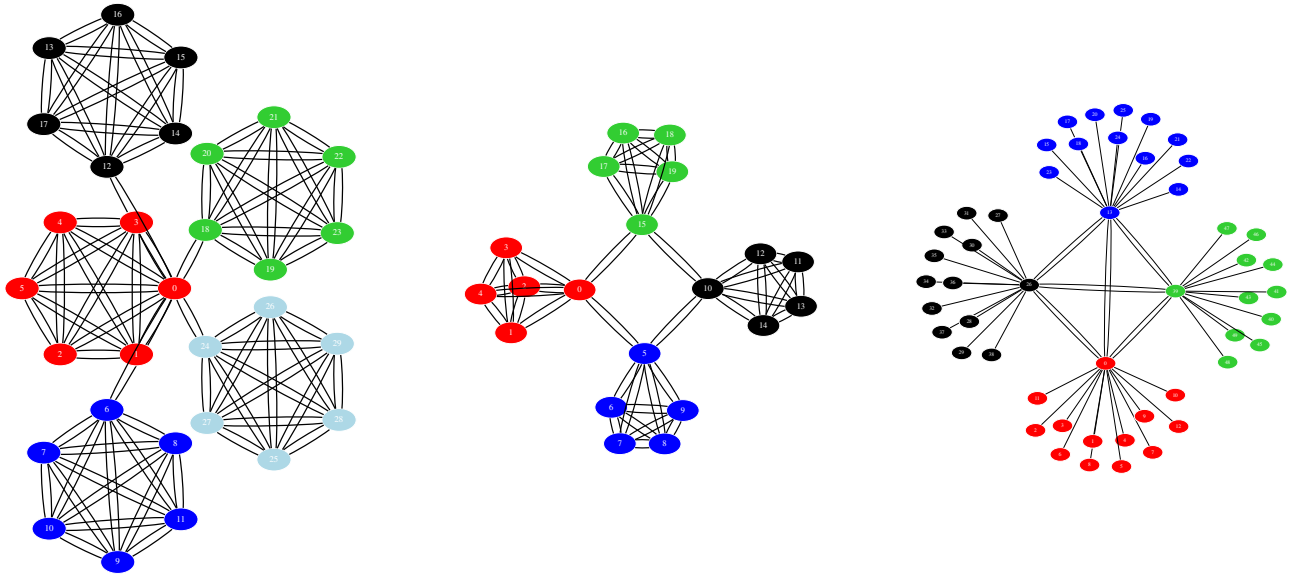
4.2 Performances

DOLMEN aims to facilitate the conception of complex distributed systems by allowing programmers to manipulate higher-level abstractions and by providing a generic framework. Both higher-level abstraction and genericity have the same usual trade-off: decreased performance. In this subsection, we show that the performances are still good, thanks to the use of tried-and-true gossip methods that are known to converge fast and to be bandwidth-efficient.

4.2.1 Convergence

The convergence toward the target topology is also pretty fast. A system of three ring-components linked together sequentially reaches a stable state in a few rounds. Initially (Figure 9a) nodes all have random neighbors, but after only two rounds (Figure 9b) the general target shape of the system is already visible, and soon a stable state is reached (Figure 9c).

All the different sub-procedures of the DOLMEN runtime, detailed in Section 3.3, follow the rapid convergence pattern



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

(b) A ring of 4 Cliques components, similar to topologies used in distributed storage

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

Figure 7: The result topologies corresponding to the previous configuration (after 10 rounds of simulation)

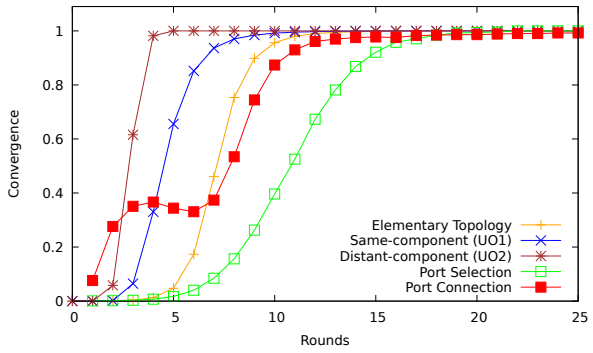


Figure 8: The different sub-procedures all follow the same rapid convergence pattern.

common in gossip protocols, as shown in Figure 8. For this experiment, we measured convergence as the proportion of nodes in the correct state regarding the target topology, from the point of view of a global omniscient observer. The drop around round 5 in *Port Connection* is due to a few nodes; they are briefly thinking they are ports (because *Port Selection* is a bit slower than the others, even if it is quite fast). Further they are connecting to distant components when they should not, before realizing they are actually not ports.

4.2.2 Dynamic Reconfiguration

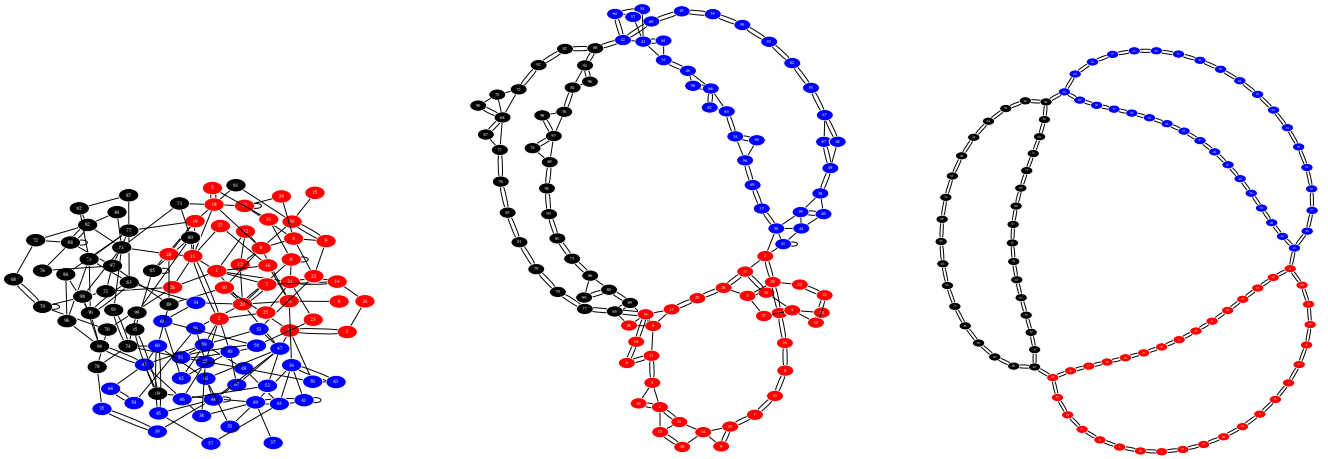
We argued that DOLMEN would help composing complex systems-of-systems and promote re-using previous works. But that means DOLMEN will need to be deployed to real systems that do not start in a random state.

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 component assignment. We shifted from a system with 3 components to 4 components, so we randomly assigned 1/3 of the nodes in each component to the new component. Many other policies may be envisioned, but due to space constraints we will only consider this one. At a given round (Figure 10b), the new configuration is sent to all the nodes, and some of them are allocated to the new component. Only 2 rounds later (Figure 10c), the nodes in the new component already found each others, and the previous components restored almost perfectly their stable state, despite losing some neighbors. Quickly, a new stable state is reached (Figure 10d). All the measures presented in the previous section 4.2.1 revealed performances as good or better for a dynamic reconfiguration from a converged system as for a system deployed from a random initial state.

4.2.3 Scalability

Our runtime's performances scale well when the number of nodes and components in the system augment. We measured the convergence time in rounds of the system for a large variety of configurations, according to the following convergence criteria:

- SameComponent overlay: at least 90% of the nodes have found 10 neighbours in the same component;
- OtherComponents overlay: at least 90% of the nodes have found a node in each component;
- CoreProtocol: at least 90% of the nodes have found their 2 closest neighbours in the ring;

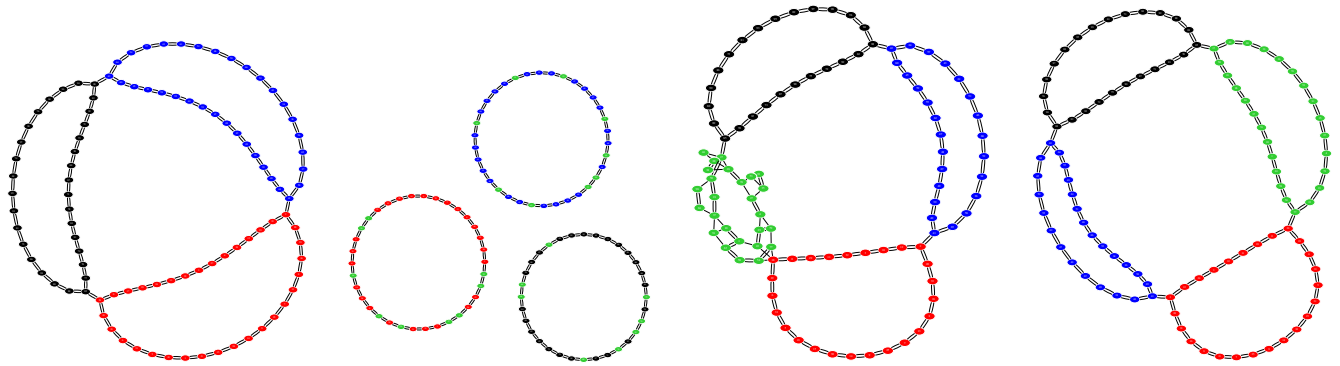


(a) Random initial state

(b) After 2 rounds, the general shape is already visible.

(c) After 6 rounds, the system has already reached a stable state.

Figure 9: Visualization of a system of 100 nodes and 3 components at different states.



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

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

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

(d) The system reaches its new stable state after 5 rounds, even quicker than with a random start.

Figure 10: Dynamic reconfiguration and convergence to a new stable state.

- PortSelection: at least 90% of the ports are assigned to the correct node (and only this one);
- PortConnection: at least 90% of the ports found their related port in the distant component.

In Figure 11, a configuration with 20 components linked together sequentially is deployed for different number of nodes.

All sub-procedures converge in a few rounds, even for large number of nodes. Most importantly, they converge as fast or faster than the Core 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 OtherComp overlay converges in constant time as the number of nodes augments. This is due to the fact that the ratio nodes/component is constant, so whatever the total number of nodes in the system, it is as likely to get a node in a given component. The abnormally high point for the SameComp overlay at 200 nodes is due to the fact that there is exactly 10 nodes per component; so the convergence criteria used means that a node must have found all other nodes in the component. But in practice, finding 6 or 7 of them is enough and does not hinder the convergence of the other sub-procedures, as depicted on the graph. For larger numbers of node per component, the convergence time is roughly constant, for the same reason as OtherComp. The other two sub-procedures scale logarithmically with the number of nodes, similar to the Core Protocol.

In Figure 12, various configurations are deployed on a system of 26500 nodes.

Convergence time increases slowly with the number of component involved in the system, and even a complex system with 20 components converges in less than 15 rounds.

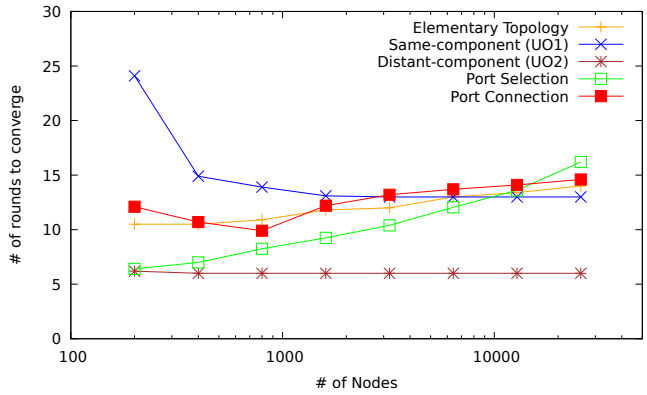


Figure 11: Convergence time of the various sub-procedures for a system of 20 components. It is fast and scales well with the number of nodes.

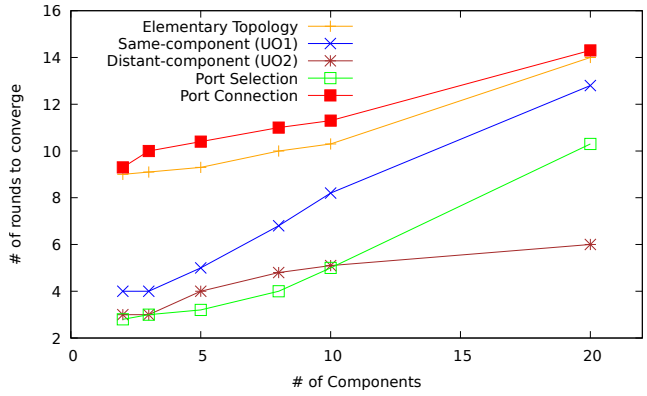


Figure 12: Convergence time of the various sub-procedures for a system of 25600 nodes. It is fast and increases slowly with the number of components.

4.2.4 *Overhead*

Compared to an ad-hoc approach optimized for a given problem, the runtime in our generalist framework incurs some 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 core protocol as our baseline.

For these measures, we considered that: a/ a node ID would use 16 bytes (IPV6 address); b/ a node "position" would use 8 bytes (64-bit double); c/ a component ID would use 8 bytes (64-bit integer).

First, Figure 13 shows that the bandwidth consumption pattern over time is similar for the baseline and the overhead. Both quickly 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 components, the bandwidth consumption per round is around 1,800 bytes, all combined.

The overhead is, of course, dependent on the complexity

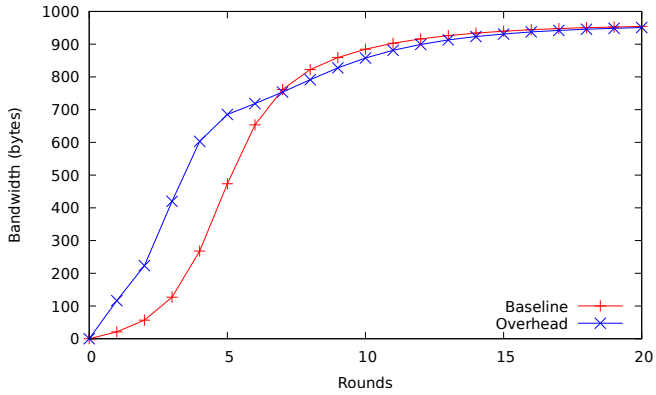


Figure 13: Comparison of bandwidth consumption (in bytes) between the core protocol and our runtime's sub-procedures, for a system of 20 components and 25,600 nodes. Both follow the same pattern, and both are very small.

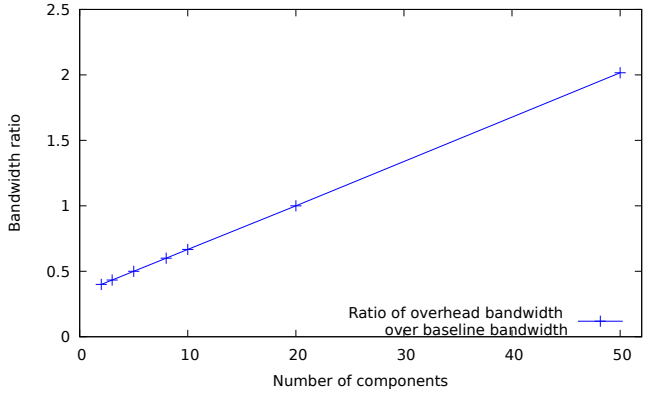


Figure 14: Relative proportion of bandwidth consumed by our runtime, compared to the base protocol, for a system of 25,600 nodes in stable state. The ratio grows linearly with the number of components and is quite small. It shows that Dolmen's overhead is very small even for 50 components (less than 2 MB in absolute value).

of the target topology. The more components and ports there are, the more messages are used to find and connect them. But even with large numbers of components, the overhead remains of a magnitude similar to the baseline. Figure 14 shows the ratio between baseline and overhead for different numbers of components 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 message exchanged are heavier and the bandwidth consumption is the highest. It increases linearly with the number of components. As depicted in Figure 14 for 50 components, the bandwidth ratio is around 2, which in absolute value represents 1900 bytes, so it represents a very negligible amount.

5. RELATED WORK

Wireless Sensor Networks have been a fertile ground for holistic programming framework aiming to simplify the programming of a very large number of distributed entities.

Among them, approaches such as Kairos [14] and Regiment [31] draw their inspiration from existing distributed programming models. They provide means to quantify over multiple nodes, and hides 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 constrains 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 supports any form of modularity or composability, one of the key aspect of DOLMEN.

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 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 composing pre-existing entities by assembly as we do.

Neighborhoods primitives such as Hood [40], Abstract Regions [39], and Logical Neighborhoods [30] are complementary to tuple spaces. They provide scoping mechanisms that limit communication to sets of nodes (regions, or neighborhood) selected according to a wide range of criteria. They are largely orthogonal to our approach, and could be exploited for instance to refine the differentiation phase of DOLMEN.

DOLMEN bears some similarity to Fragmented Objects [18, 22], in which a component's state is distributed (fragmented) among a number of distrib-uted 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 mechanism, they can allow a component to extend or retract according to current systems needs. However, implementations of fragmented components proposed so far [18] tend to be heavy-weight. They also typically rely solely on RPC, an interaction paradigm that is ill-suited to loosely coupled large-scale systems.

6. FUTURE WORKS & CONCLUSION

Large scale distributed systems are becoming omnipresent, and are, at the same time, increasingly complex. It becomes a particularly tiresome and cumbersome task for developers to specify and implement such systems. From the last decades, lots of efforts have been done to ease their development. However most of the focus has been on the local behavior of individual nodes rather than on the programmatic means to describe a system's global structure and behavior.

To address this challenge, we have introduced the DOLMEN framework. DOLMEN is based, on one hand, on software engineering design principles such as component-based programing, and on the other hand, on self-organizing overlays. However, DOLMEN goes a step forward by considering components as collective distributed entities and by enabling the creation of resilient, scalable, and complex distributed topologies through the assembly of components.

To reach this aim, the DOLMEN framework comprises: (i) a DSL, (ii) a component library, and (iii) a runtime. The DOLMEN DSL enables developers to write a DOLMEN configuration file that describes a complex distributed topology in an easy manner: mainly by specifying the components they want to link. The DOLMEN component library provides a set of elementary topology that developers can pick up. Finally, once the DOLMEN configuration is compiled and deployed, the runtime of each node processes several sub-procedures, to overcome various challenges such as dynamically identifying nodes that belongs to a components, creating adequate links among components as well as maintaining and handling on the fly their communications.

Further, we have demonstrated that our approach enables to create in an efficient and scalable way an array of different topologies, representative of real-world applications, that would have been difficult to realize otherwise. In particular, we have lead a thorough evaluation from four different perspectives: convergence, dynamicity, scalability, and overhead. As a result, it appears that large scale distributed systems built with DOLMEN: (i) are able to converge fastly toward the target topology expected by the developer, (ii) are converging quickly to a stable state when dynamic reconfigurations occur, (iii) are scaling well with the increasing number of nodes, and finally (iv) have a negligible overhead. As future works, we plan to extend our approach with various toolchains to deploy our systems to physical resources.

7. 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. *Journal of Computer and System Sciences*, 79(2):291–308, 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: Proceedings of the 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 *Distributed Computing Systems (ICDCS), 2014 IEEE 34th International Conference on*, 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. *Software: Practice ...*, pages 1257–1284, 2006.

[7] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes. Borg, omega, and kubernetes. *Communications of the ACM*, 59(5):50–57, 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] G. Coulson, G. Blair, P. Grace, F. Taiani, A. Joolia, K. Lee, J. Ueyama, and T. Sivaharan. A generic component model for building systems software. *ACM TOCS*, 26(1).

[11] A. Deshpande and S. Madden. Mauvedb: supporting model-based user views in database systems. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 73–84, New York, NY, USA, 2006. ACM.

[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)*, number 3560 in LNCS, pages 126 –140, 2005.

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

[16] M. Jelasity, A. Montresor, and O. Babaoglu. T-Man: Gossip-based fast overlay topology construction. *Computer Networks*, 53(13):2321–2339, Aug. 2009.

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

[18] 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.

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

[20] J. C. A. Leitao and L. E. T. Rodrigues. Overnesia: A resilient overlay network for virtual super-peers. In *SRDS*, 2014.

[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] H. Miranda, A. Pinto, and L. Rodrigues. Appia, a flexible protocol kernel supporting multiple coordinated channels. In *in Proc. 21st Int. Conf. on Dis. Comp. Sys. (ICDCS-21)*, pages 707–710. IEEE, 2001.

[27] 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/.

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

[29] A. Montresor, M. Jelasity, and O. Babaoglu. Chord on demand. In *Proc. of the IEEE Int. Conf. on Peer-to-Peer Comp (P2P'05)*, pages 87–94. IEEE, August/September 2005.

[30] 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.

[31] 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.

[32] L. Seinturier, P. Merle, R. Rouvoy, D. Romero, V. Schiavoni, and J.-B. Stefani. A component-based middleware platform for reconfigurable service-oriented architectures. *Software: Practice and Experience*, pages n/a–n/a, 2011.

[33] 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.

[34] 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/.

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

[36] R. van Renesse, K. Birman, M. Hayden, A. Vaysburd, and D. Karr. Building adaptive systems using ensemble. *Softw. Prac. and Exp.*, 28(9):963–979, 1998.

[37] 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.

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

[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.