Towards a holistic construction of opportunistic large-scale distributed systems

Simon Bouget

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE RENNES 1

THÈSE DE DOCTORAT DE

L'UNIVERSITE DE RENNES 1
COMUE UNIVERSITÉ BRETAGNE LOIRE

ECOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : Informatique

Par

Simon BOUGET

Towards a holistic construction of opportunistic large-scale distributed systems

Thèse soutenue à IRISA / Inria Rennes – Bretagne Atlantique le 20 septembre 2018
Unité de recherche : IRISA (UMR 6074) – Project-Team WIDE

Rapporteurs avant soutenance :

Philippe LAMARRE	Professeur, INSA Lyon, LIRIS (BD)
Paulo FERREIRA	Associated Professor, University of Oslo (PSE)

Composition du Jury :

Président :	Olivier BARAIS	Professeur, Université de Rennes 1, IRISA / Inria (DiverSE), B<>COM
Examinateurs :	Philippe LAMARRE	Professeur, LIRIS / INSA Lyon (BD)
	Paulo FERREIRA	Associated Professor, University of Oslo (PSE)
	Sonia BEN MOKHTAR	Directeure de recherche, LIRIS / CNRS (DRIM)
Dir. de thèse :	Francois TAIANI	Professeur, Université de Rennes 1 (ESIR), Inria (WIDE)
Co-dir. de thèse :	Yérom David BROMBERG	Professeur, Université de Rennes 1 (ESIR), Inria (WIDE)

Invité :

Étienne RIVIERE	Professeur, UCLouvain, École Polytechnique de Louvain

Titre : Vers une construction holistique des systèmes distribués opportunistes à large échelle

Mots clés : approches holistiques ; systèmes opportunistes ; protocoles épidémiques ; systèmes distribués ; large échelle ; Internet des Objets

Résumé : Avec le développement de l'*IoT*, des *Smart Cities*, et d'autres systèmes large-échelle extrêmement hétérogènes, les systèmes distribués deviennent à la fois plus complexes et plus omniprésents de jour en jour et seront bientôt difficiles à gérer avec les approches courantes. Pour masquer cette difficulté croissante et faciliter leur gestion à tous les stages de leur cycle de vie, cette thèse soutient qu'une approche holistique est nécessaire, où la fonction d'un système est considérée comme un tout, et qui se détache du comportement des composants individuels. En parallèle à cette montée en abstraction, les blocs de base doivent devenir plus autonomes, capable de réagir aux circonstances et d'automatiser la plupart des tâches de bas niveau.

Nous proposons trois contributions vers cette vision: 1) **Pleiades** : une approche holistique pour construire des structures complexes par assemblage, facile à programmer et soutenue par un moteur d'exécution auto-organisant et efficace, basé sur des protocoles épidémiques. 2) **Mind-the-Gap** : un protocole épidémique de détection des partitions dans les MANETs, grâce à des agrégations opportunistes et à une représentation stochastique compacte du réseau. 3) **HyFN**: une extension des protocoles épidémiques traditionnels, capable de résoudre efficacement le problème des k-plus-*lointains*-voisins, ce dont les méthodes standards s'étaient révélées incapables jusqu'à maintenant.
Nous considérons que ces trois contributions montrent que notre vision est réaliste, et mettent en valeur ses qualités.

Title: Towards a holistic construction of opportunistic large-scale distributed systems

Keywords: holistic approaches; opportunistic systems; epidemic protocols; distributed systems; large scale; Internet of Things

Abstract: With the advent of the IoT, Smart Cities, and other large-scale extremely heterogeneous systems, distributed systems are becoming both more complex and pervasive every day and will soon be intractable with current approaches. To hide this growing complexity and facilitate the management of distributed systems at all stages of their life-cycle, this thesis argues for a holistic approach, where the function of a system is considered as a whole, moving away from the behavior of individual components. In parallel to this rise in abstraction levels, basic building blocks need to become more autonomous and able to react to circumstances, to alleviate developers and automate most of the low level operations.

We propose three contributions towards this vision: 1) **Pleiades:** a holistic approach to build complex structures by assembly, easily programmable and supported by an efficient, self-organizing gossip-based run-time engine. 2) **Mind-the-Gap:** a gossip-based protocol to detect partitions and other large connectivity changes in MANETs, thanks to periodic opportunistic aggregations and a stochastic representation of the network membership. 3) **HyFN:** an extension to traditional gossip protocols that is able to efficiently solve the k-*Furthest*-Neighbors problem, which standard methods have been unable to up to now.
We believe these three contributions demonstrate our vision is realistic and highlight its attractive qualities.

Résumé substantiel en Français

Contexte

Les systèmes distribués à large échelle ont désormais pénétré notre vie de tous les jours. Ces systèmes sont depuis longtemps une composante clé de l'Internet, fournissant une large gamme de services, des traditionnelles applications web en ligne comme les moteurs de recherche (e.g. Google) ou les réseaux sociaux (Facebook, Twitter, ...), au partage de fichier pair-à-pair (e.g. Bittorrent) ou plus récemment aux crypto-monnaies. Avec l'avènement de l'Internet des Objets (ou objets connectés, *IoT*, pour *Internet of Things*), cette tendance s'est encore accélérée, avec des systèmes distribués désormais étroitement mêlés à notre environnement physique. Cette évolution nous laisse envisager un avenir dans lequel les utilisateurs seront entourés par un nombre croissant d'appareils connectés, travaillant de manière autonome vers leurs objectifs individuels tout en cherchant à coopérer et à exploiter les opportunités offertes par leurs voisins. Que ce soit à leur domicile, au travail ou sur la route, les humains auront régulièrement à interagir avec des systèmes distribués dans leur vie courante.

Au fur et à mesure que ces systèmes sont de plus en plus répandu, ils croissent également en taille et en complexité. Par exemple, une installation domotique typique se compose de dizaines de capteurs (pour la lumière, la température, la détection de mouvement, répartis dans différentes pièces, etc.) mais aussi d'actionneurs qui contrôlent les différents éléments d'une maison. Ce type d'installation repose également sur le *Cloud* pour relayer les informations à un utilisateur distant, pour accéder à des statistiques ou à des protocoles de décision utilisant de l'apprentissage automatique. En d'autre termes : les systèmes distribués modernes incluent non seulement un plus grand nombre d'appareils, mais aussi une plus large gamme d'appareils et de services très différents et hétérogènes.

Problématique

Leur complexité croissante rend les systèmes distribués modernes particulièrement difficiles et coûteux à développer et maintenir. Ils deviennent rapidement trop larges, impliquant trop de composants différents, pour être appréhendé par un individu isolé. Cette limitation cognitive a plusieurs conséquences négatives : le processus de développe-

ment devient plus vulnérable aux erreurs humaines, le système est délicat à configurer, déployer et maintenir efficacement, la surveillance et le provisionnement sont extrêmement chronophages en raison du nombre astronomique d'évènements générés, sans même tenir du compte de la complexité croissante de chaque évènement individuel.

Autrement dit, on observe une tension croissante entre la nécessité de prendre du recul pour être capable d'appréhender un système dans son ensemble, et le besoin de gérer en détail des comportements individuels de plus en plus complexes. Plus précisément, nous avons identifiés quatre challenges majeurs.

- Se concentrer sur la vue d'ensemble du système, pas le comportement individuel de chaque composant. Toutefois, ceci est difficile avec les approches de développement traditionnelles qui consistent précisément à implémenter le comportement individuel de chaque appareil plutôt que la fonction globale du système.
- Déployer et maintenir un système opérationnel avec un très grand nombre de composants. Comment configurer chaque composant individuel, comment amorcer le système ? De plus, même après un déploiement réussi, la taille du système garantit que des plantages vont inéluctablement se produire : au moins l'un des composants, par la simple force du nombre, subira une défaillance.
- Rendre les systèmes capables de réagir à des circonstances changeantes. En pratique, aucune configuration ne peut convenir à toutes les circonstances qu'un système opérationnel rencontrera, mais ajuster les réglages d'un système large échelle en production est extrêmement complexe. Le nombre d'actions distinctes à effectuer peut être très grand, des conflits inattendus peuvent survenir, ou des dépendances implicites peuvent être mises en défaut.
- Rendre les systèmes capable d'évoluer au cours du temps. En effet, tout système déployé pour une période significative verra son environnement se modifier, avec de nouvelles infrastructures, de nouvelles fonctionnalités à exploiter et de nouveaux besoins à satisfaire. Un système performant devrait être capable d'évoluer en parallèle avec son environnement, mais mettre en place ce type d'évolutivité pour des changements qui sont inconnus au moment de la conception peut être particulièrement délicat.

Notre vision

Pour répondre à cette tension entre le besoin d'une vision globale et la nécessité de gérer des détails complexes, et pour adresser les challenges ci-dessus, notre intuition est qu'une approche double est nécessaire : nous devons fournir aux concepteurs et développeurs une **approche holistique** qui permet de mettre en place des systèmes distribués **opportunistes**.

Contributions

Cette thèse avance trois contributions vers la vision que nous défendons.

Pleiades

Pleiades est un *framework* permettant de construire dans un réseau distribué des structures complexes sans aucun élément coordinateur. Il est composé de trois grandes parties, un langage domaine, une bibliothèque de formes élémentaire et un moteur d'exécution générique reposant sur une combinaisons de protocoles épidémiques auto-organisants qui résistent extrêmement bien en cas de défaillance et sont facilement reconfigurables. Ce framework permet de concevoir un grand nombre de structures complexes avec des fichiers de configuration simples et expressifs, par assemblage de formes élémentaires issues de la bibliothèque.

MtG

MtG est un protocole distribué de détection de partition dans les réseaux mobiles ad-hoc. En utilisant une structure de donnée probabiliste et extrêmement compacte, il arrive à détecter pratiquement toutes les partitions apparaissant dans un réseau ad-hoc avec un très faible taux d'erreurs, y compris dans des circonstances difficiles avec un taux de perte des messages élevé.

HyFN

Finalement, HyFN est un algorithme distribué de construction de graphe des k-plus-lointains-voisins, un problème complémentaire aux k-plus-proches-voisins mais que les méthodes actuelles ne permettent pas de résoudre efficacement. Se reposant sur des protocoles épidémiques et une approche à deux couches, il est efficace et montre que des combinaisons simple de protocoles traditionnels permettent malgré tout de résoudre des systèmes complexes.

Conclusion

Ces différentes contributions reposent sur des modèles théoriques validés par des simulations et des analyses mathématiques détaillées. Bien qu'elles ne permettent pas de réaliser l'intégralité de la vision que nous défendons, elles en démontrent la pertinence et la faisabilité.

Vers des approches holistiques

PLEIADES, notre framework holistique pour créer des topologies complexes, montre que des abstractions adaptées permettent de concevoir facilement une large gamme de systèmes complexes. Il ne permet pas encore de mettre au point tout type de système distribué et se concentre spécifiquement sur la structure des systèmes, mais il ouvre des pistes intéressantes, telles que le développement d'une interface standardisée entre la structure sous-jacente d'un système et l'application qui est mise en œuvre par dessus.

Vers des systèmes opportunistes

PLEIADES réalise des fonctions complexes de maintenance et de résilience en combinant plusieurs protocoles épidémiques isolément simples. C'est un signe encourageant que les méthodes épidémiques sont effectivement adaptées pour créer des systèmes opportunistes, capables de réagir à leur environnement sans intervention explicite des opérateurs humains.

MtG permet de détecter de manière relativement fiable et extrêmement efficace une certaine classe d'évènements se produisant dans les réseaux mobiles ad-hoc, les partitions et autres changements notables de connectivité. C'est un premier pas vers des systèmes capables de s'auto-adapter, et une démonstration convaincante des performances des méthodes épidémiques même dans des circonstances difficiles. Les recherches doivent toutefois se poursuivre dans deux directions indépendantes :

- d'une part, détecter plus de types d'évènements, avec une granularité plus faible, et dans des contextes plus larges qu'uniquement les réseaux mobiles.
- d'autre part passer de la simple détection à l'adaptation, avec des systèmes capables d'analyser les informations qu'ils détectent et de prendre des décisions vers un nouvel objectif mis à jour.

HyFN, en étendant le champ d'application des méthodes épidémiques, nous convainc qu'elles constituent un outil adapté au développement de solutions génériques et de systèmes auto-organisants, capable de s'adapter à des circonstances très diverses, et d'évoluer progressivement au cours de leur cycle de vie, en même temps que leur environnement.

Globalement, l'ensemble de nos contributions montrent que notre vision, une approche holistique pour construire des systèmes opportunistes composés de blocs génériques et auto-organisants, est viable. Elles ouvrent également de futures directions de recherche pour poursuivre le travail vers cette vision.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Context

Large-scale distributed systems have come to pervade our everyday lives. These systems have long been part of the Internet, providing a large range of services including traditional on-line web applications such a search or social networks, Peer-to-Peer file sharing, or more recently cryptocurrencies. The advent of the Internet of Things (IoT) has further accelerated their rise, by closely embedding distributed systems into the physical world. This evolution lets us envisage a future in which users will be surrounded with an increasing number of connected devices working autonomously toward their individual goals, seeking to cooperate and leverage on each other. Whether it be at home, at work, on the road, humans will have to interact with distributed systems in their day-to-day life.

Take, for instance, the recent progresses toward Self-Driving Cars. Autonomous fleets of cars will very likely need to communicate with each other as well as with their surrounding environment, in order to obtain key information such as weather and traffic conditions, emergency announcements, and city regulations. Similarly, buildings are becoming smarter thanks to the generalization of home automation techniques, which rely on distributed systems to control essential elements such as lighting, air conditioning, entertainment systems, or building security. Increasingly, smart buildings are expected to connect with each other, and with city information systems, in order to form what has been termed *Smart Cities*. As envisioned, smart cities will be able to collaborate towards higher level goals such as managing assets (infrastructure) and resources (water, energy, ...) more efficiently within large urban areas.

As the above systems become more common, they also grow in size and complexity. A typical Home Automation set-up contains dozens of sensors (for light, temperature, motion-detection) but also actuators to control the different elements of a home. It also relies on the Cloud to relay and access statistics or machine-learning-assisted decision making. In other words, modern distributed systems not only include more devices, they also include a wider range of device types and functionality.

The resulting complexity naturally rises over time as systems evolve in order to adapt to evolving demands and a changing technological landscape. This is because a distributed system cannot be built in an initial ideal state and simply frozen for the rest of its lifetime. As the system is used, new resources are added, new infrastructure is deployed, new sub-systems and features are added to the whole, new nodes join, potentially equipped with a newer technology, and all these changes require in turn the system to adapt and transform itself.

Finally, modern distributed systems have to satisfy a number of hard requirements under changing circumstances: workload may vary by orders of magnitude with the time of day or seasons passing by; devices are likely to be leaving and joining the network on a continuous basis, either because of failures (datacenters) or of willing cessations in participation (P2P content sharing); hard constraints must be met in terms of latency (Self-Driving Cars). Adding in measures to guarantee those requirements participates in the rising complexity of modern-days distributed systems.

1.2 Challenges for modern distributed systems

Their growing complexity is making modern distributed systems particularly difficult and costly to develop and maintain. Modern systems are rapidly becoming too large, too complex, they are involving too many different components to be fully grasped by a single individual. This cognitive limitation has a number of negative consequences for their development.

When designing new features and applications, development teams must keep in mind all the various components of a complex system, all their multifaceted relationships. As the complexity of a distributed system increases, this process becomes more error-prone, the system becomes more difficult to configure, deploy, and maintain efficiently.

This difficulty continues on as the system goes in production, and must be monitored and provisioned. Without proper tools, the sheer scale of modern systems means it is extremely costly for an operator to react to every single event in order to adapt to changing circumstances or to correct arising issues with an appropriate action.

In both cases, the core issue is a difficulty to consider the system as a whole, without getting stuck into details. More precisely, we identify three key challenges:

Challenge #1: Focus on the function of the system, not the behavior of individual component For a distributed storage system, for instance, what is important is that each element is correctly stored, with an appropriate level of redundancy, and is easily retrievable with good performances and availability. The node-to-node low-level communications on the other hand, while ensuring these desired properties, do not represent the real added value of the system. Yet this holistic view is difficult to maintain within a traditional development process that focuses on implementing the

behavior of individual node or class of nodes, rather than on the high-level functions delivered by the distributed system as a whole.

Challenge #2: Deploy and maintain a live system with a very large number of components Deploying a large-scale system is a complex task: how do you properly configure each individual component? How do you boot-strap? Moreover, even after a successful initial deployment, large-scale systems are highly likely to experience some crashes or other run-time issues, simply due to the large number of components involved: one of them is bound to have a problem. As a consequence of all that, managing a large-scale distributed system and keeping it operational can be daunting.

Challenge #3: Make system able to react to changing circumstances and to evolve over time In realistic situations, no single configuration is appropriate to all circumstances, but tweaking the configuration of a large, live system is extremely complex: the number of separate actions needed can be very large, unexpected conflicts may arise, or unforeseen dependencies. Also, due to the heterogeneous context mentioned above, systems deployed for any significant length of time will see their environment evolve around them, new infrastructure, new features. A good system should be able to evolve in parallel, to adapt to and leverage its new environment, but building-in this kind of forward-looking adaptability to evolutions that are still unknown can be extremely tricky.

1.3 Our vision: opportunistic systems with a holistic approach

In order to address the above challenges, we posit that the current ecosystem needs to adopt **a more holistic and high-level approach**, at all stages of a system life, combined with **opportunistic basic building blocks** able to collaborate and self-adapt to evolving circumstances. We argue that high-level abstractions are needed to better design systems as a whole, while focusing on desired features and properties, rather than on low-level behaviors. These abstractions should ideally be embodied within generic frameworks and smart run-time platforms with the ability to automate most of the low level work, in order to streamline development, deployment and maintenance efforts.

At this point, it is interesting to stop and remark that this is similar to the process other fields of computer sciences have already gone through, such as Programming Language Theory or Software Engineering. Things start with some ad-hoc solutions to a few problems, then they are extended, generalized, theories are formalized, standards emerge and best-practices are put in place. This is generally accompanied by a rise in abstraction levels, with more recent iterations hiding a large part of the complexity and

letting the machines handle the details. Just like Programming Languages went from Assembly to C to high-level programming paradigms such as Functional or Object-Oriented, and now let compilers do most of the optimizations, or garbage collectors do most of the memory management, large-scale modern distributed systems need to go through a similar evolution.

Leveraging this parallel, we have sought in this work to organize our thought process along the following two lines:

- we have sought to study the best practices put in place in other fields, and where possible import and adapt them to distributed systems;
- we have then endeavored to provide new tools in order to tackle the challenges specific to distributed systems, while hiding most of their inherent complexity.

Best practices from other fields The key point, as we already mentioned, is a rise in abstraction levels. This provides a better mental framework to design more complex system without getting lost among the hundreds or thousands individual low-level components of a modern distributed system. Another notable evolution is a push toward compartmentalization, that is to say making sure that each component in a system fulfills a single function and is properly isolated from other components that fill different functions. In turn, this promotes reusability and modularity, enabling the same work to be used in multiple places, or to change some small isolated part of a system without breaking everything. Interestingly, a push toward compartmentalization is already in effect for distributed systems, with trends such as containers and micro-services, and is part of the reason why the complexity of modern systems is increasing. But without an accompanying rise in abstraction levels, the resulting complexity will quickly become unmanageable. Finally, the last important aspect is encapsulating common low-level use-cases in generic parts. This has a number of positive impacts, such as hiding complexity (just like better abstractions, but from the other side of the question), or more optimized code that benefit the whole community. But what would be such low-level components in the context of distributed system? That is what we will see now.

Specific issues with distributed systems Very promising candidates on this side are self-organizing overlays and epidemic protocols (also called gossip protocols). They are a family of protocols that naturally possess a number of desirable properties: fully decentralized, highly resilient, very efficient both in terms of speed and bandwidth consumption. Working with successive greedy, local optimizations, they manage to gather or disseminate information all over a distributed system extremely quickly.

Using them as a basis, it is possible to build more advanced protocols that, through periodic gathering of information, monitor the state of a system, detect changing circumstances, adapt to them and propagate new updated information to other participants. Those changing circumstances can be crashes or failures (since self-organizing

overlays are naturally resistant to it), but also additional resources being added to the system, new information from sensors, other sub-systems available or even third-party willing to cooperate. Leveraging epidemic protocols, it is thus possible to greatly simplify the deployment and maintenance of distributed systems, effectively hiding a lot of the complexity from the system operators.

In conclusion, by combining: (i) a rise in abstraction levels and adoption of best-practices from other fields; with (ii) self-organizing overlays and gossip to enable distributed systems to monitor themselves, detect changes and adapt to their environment, we propose in this thesis to progress towards a **holistic approach for the development of opportunistic systems**.

1.4 Contributions

More precisely, we make the following three contributions in order to get closer to this vision:

A holistic framework for complex topologies First off, we propose PLEIADES, a framework to build, deploy and maintain complex network topologies seen as an assemblage of simpler shapes. In the specific context of network overlays, PLEIADES addresses **Challenge #1** by providing high-level abstractions to design complex topologies while ignoring the individual behaviors of each nodes, and **Challenge #2** thanks to a simple and efficient run-time engine based on epidemic protocols, which is able to automatically maintain and repair the system to preserve the desired topology in most circumstances, without any direct action from an operator.

This contribution is based on the work that has been presented in the following paper:

- Simon Bouget, Yérom-David Bromberg, Adrien Luxey, François Taïani: Pleiades: Distributed Structural Invariants at Scale. In the *48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2018, Luxembourg City, Luxembourg, June 25-28, 2018. Proceedings*, 2018 [14]

A partition-detection protocol for MANETs Our second contribution, MIND-THE-GAP (MtG for short), addresses the first step of **Challenge #3** in the context of Mobile Ad-hoc Networks (MANETs). MtG is an epidemic protocol which enables MANETs to detect some kind of changes in their environment, namely partition events and other large change in network connectivity, in order to react and adapt to it. Furthermore, MtG was designed with the advent of opportunistic systems in mind and proposes two variants: a self-detection method, and an assisted detection method which is able to leverage the presence of third-party systems in the environment.

This contribution is based on the work that has been presented in the following paper:

- Simon Bouget, Yérom-David Bromberg, Hugues Mercier, Etienne Rivière and Francois Taiani: Mind the Gap: Autonomous Detection of Partitioned MANET Systems using Opportunistic Aggregation. In the *37th IEEE International Symposium on Reliable Distributed Systems, SRDS 2018, Salvador, Bahia, Brazil, October 2-5, 2018. Proceedings,* 2018 [15]

An extension to traditional epidemic protocols Our third contribution comes from a simple observation: standard epidemic protocols suffer from an important restriction, they only work efficiently if there is some pseudo-transitivity between nodes, some regularity that allows iterative greedy optimizations to function properly without getting stuck in sub-optimal states.

However, (i) this hypothesis is unlikely to hold when a system tries to find complementary nodes to collaborate on a common task, because complementary nodes are as different as possible, not similar at all; **and** (ii) finding complementary nodes that can collaborate is one of the key tenets of opportunistic systems, so this problem is almost guarantee to arise sooner rather than later. Hence why we decided to explore if we could lift that restriction and we finally propose HyFN (for Hybrid Further Neighbors), a two-layered epidemic protocol with good performances and low cost even in the absence of any transitivity. We thus demonstrate the wide applicability of epidemic protocols, and that they are great candidates to further address the many facets of **Challenge #3** in a more general fashion.

This contribution is based on the work that has been presented in the following paper:

- Simon Bouget, Yérom-David Bromberg, François Taïani, Anthony Ventresque: Scalable Anti-KNN: Decentralized Computation of k-Furthest-Neighbor Graphs with HyFN. In *Distributed Applications and Interoperable Systems, DAIS 2017*, held as part of the *12th International Federated Conference on Distributed Computing Techniques, DisCoTec 2017, Neuchâtel, Switzerland, June 19-22, 2017. Proceedings*, pages 101-114, 2017 [16]

1.5 Document organization

This dissertation is organized as follows. Chapter 2 presents the background and state of the art in more details. Chapters 3 to 5 describe our three main contributions, one in each chapter: Chapter 3 focuses on our holistic approach for complex topologies (Pleiades); Chapter 4 moves on to our partition detection protocol (Mind-the-Gap); and Chapter 5 deals with our extension of epidemic protocols to

Further Neighbors (HyFN). Finally, Chapter 6 presents our conclusions and possible new research directions opened by our work.

Chapter 2

State of the Art

In the previous chapter, we argued that distributed systems need to evolve toward a holistic approach with high-level abstractions and opportunistic systems able to react autonomously and adapt to evolving circumstances.

This chapter will thus be organized along two main axis. First, in Section 2.1, we focus on abstractions, we present the holistic approaches used in other fields, how distributed systems are currently developed, and what programming models exist for distributed systems. Then, in Section 2.2, we present the various mechanisms designed to make distributed systems more aware of their environment and able to react to circumstances. Finally in Section 2.3, we sum up our findings, comment on the overall technological landscape and highlight what is still missing to realize the vision we proposed.

2.1 Programming abstractions and Holistic Approaches

In this Section, we focus on how systems are developed and deployed. We start with the abstractions used for non-distributed systems (Subsection 2.1.1), then we move on to frameworks used in static distributed systems, where nodes don't move and usually use wired connections and are able to contact any other node (Subsection 2.1.2), and finally we discuss the approaches used in wireless and mobile networks, where nodes are limited by their range of communication and usually have more constrained resources, and where the systems are much more dynamic (Subsection 2.1.3).

2.1.1 Non-distributed systems

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 [20, 27].

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.

2.1.2 Static distributed systems

There is currently a trend to combine various services to realize more complex features. It has been popularized especially 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 [74, 57]. 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.

Originally proposed in the context of fixed networks [33], 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 [26], and TOTA (Tuple On The Air) [55]. Interestingly, TOTA moves away from nodes as a key programming abstraction and focus on messages instead, which act as lightweight agents. TOTA messages carry a representation of their own behavior in terms of production of markers (akin to pheromones) and attraction rules: the messages become the active entities which are programmed, while nodes simply offer a medium in which these evolve. TOTA is thus related to alternative computation models such as chemical programming, in which computations arise from the asynchronous reaction of (symbolic) molecules in a chemical solution [4].

Neighborhoods primitives such as Hood [81], Abstract Regions [80], and Logical Neighborhoods [63] 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 the approach we argued for in the previous chapter.

In contrast to macro-programming techniques, both tuple-based approaches and neighborhood primitives tend to encourage a loosely coupled, decentralized view of a wireless distributed system. In this view, programmers have the ability to finely code

the behavior of localized entities (nodes, messages, or neighborhoods), but they were not designed to provide modular and easily composable entities.

2.1.3 Wireless and mobile networks

Wireless Sensor Networks have been a fertile ground for holistic programming framework. These approaches seek to alleviate the task of developers by offering higher-level programming approaches that expose a WSN as one single programmable entity. These works differ in the extend to which they reify underlying nodes in their programming model.

Among them, approaches such as Kairos [37] and Regiment [65] draw their inspiration from existing distributed programming models. Kairos [37] relies on a shared memory model with distributive constructs similar to those of parallel programming languages, while Regiment [65] uses functional programming and builds on the concepts of streams and aggregation. They share however the same fundamental traits: 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 [29, 53, 13]. 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, and represent a major advance over low level execution frameworks. However, Kairos and Regiment still can be challenging to wrestle with, as they use complex macro-operations to capture distribution. TinyDB and Cougar on the other hand take an almost exclusively data-oriented view of WSNs, which limit their applicability to richer scenario. More generally, and perhaps not surprisingly since this was not their primary intent, none of them support strong modularity. In particular, they do not allow developers to easily express interactions between reusable software entities, or to reason explicitly about dependencies, a critical enabler to build complex systems from reusable components.

2.2 Mechanisms of adaptation

We already mentioned epidemic protocols in Chapter 1 as very promising candidates to build opportunistic systems, so we start this section by examining them in more details, especially their use in self-organizing overlays (Subsection 2.2.1). We then move on to a number of more narrow problems and constraints that modern distributed systems have to tackle, and the various adaptation mechanisms that have been developed

to solve them: Scalability, or the ability to maintain good performances even when the number of elements in a system increases greatly (Subsection 2.2.2); Partition Detection, or being able to realize when a network get split in two, a pressing concern in Mobile Ad-Hoc Networks, (Subsection 2.2.3).

2.2.1 Self-organizing overlays

Gossip protocols are a family of distributed protocols that disseminate information in a computer network in a similar way to rumors propagating in a society. They are also called *epidemic protocols*, in reference to how diseases spread. Each node initially only knows about a small number of neighbors, representing its *local view* of the network. Thanks to periodic exchange with those neighbors however, information quickly spreads all over the network. There are a lot of variants, with push, pull and push-pull models, and a varying use of random information to quicken the process, but due to their fully decentralized nature, gossip protocols in general are highly scalable, quick, robust and use relatively little bandwidth when compared to more brute force approaches like flooding.

One of the big applications of gossip protocols are self-organizing overlays [42, 79], a family of decentralized protocols that are able to autonomously arrange a large number of nodes into a predefined topology (e.g. a torus, a ring). The idea is that with every exchange of information, a node can use the new data to greedily update its neighborhood and improve its view of the network until it manages to reach the predefined topology. Self-organizing overlays are just as fast as the gossip protocols they are based upon, resilient and self-healing, and can with appropriate extension, conserve their overall shape even in the face of catastrophic failures [17]. They can be used to create a ring overlay (Pastry [68]), a Euclidean space (CAN [66]), or even a random graph (Random-Peer-Sampling [43]). Some approaches, such as Vicinity [78] and T-Man [42], can even use a configurable distance function that enables them to build a wide range of shapes, based on the distance considered.

Another domain of application for gossip protocols with a large body of works is the decentralized construction of k-nearest-neighbors graphs (KNN). In such systems, nodes (e.g. representing a user) can connect to each other using point-to-point networking, but once again only maintain a small partial view of the rest of the system, typically a small-size neighborhood of other nodes. Each node also stores a profile (e.g. a user's browsing history, or the movies they liked), and uses a peer-to-peer epidemic protocol to greedily converge towards an optimal neighborhood, i.e. a neighborhood containing the k most similar other nodes in the system according to some ranking function on profiles (e.g. cosine similarity, or Jaccard's coefficient). Some variants also evaluate a neighborhood as a whole, so a set of nodes in one go, instead of ranking profiles node by node [10]. KNN construction usually involves profile spaces with much higher dimensionality but a much less regular structure than self-organizing overlays,

but they work similarly, with the ranking function taking the role of the distance function.

Both applications have been shown to work in a large variety of settings [3, **?**, **?**]. Notably, the scalability and robustness of these solutions have made them particularly well adapted to large scale systems such as decentralized social networks [56], recommendation engines [5, 32, 47], news dissemination [18], search optimization [31], and peer-to-peer storage systems [24].

Self-organizing overlays such as T-Man or Vicinity 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.

2.2.2 Scalability

Scalability is the ability for a system to maintain good performances even when the number of users increases by orders of magnitude. With modern applications like file-sharing, video-streaming, or social network involving tens of thousands to billions of users, this is an increasingly important property for modern systems. It can be achieved in many different ways, including the aforementioned gossip protocols which, since they only act locally in the network, don't care much about the total size of the system and thus remain highly scalable.

Another notable and historical approach is the concept of Fragmented Objects [54], 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 mechanism, they can allow a component to extend or retract according to current systems needs. However, implementations of fragmented components proposed so far [44] 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.

2.2.3 Partition detection

Partitions are often highly problematic to the workings of MANETs and Wireless Sensor Networks, and have therefore been investigated in the past. [1]

Membership and partition detection The work of Arantes *et al.* [25, 2] formalizes the notions of partition detector and partition participants detector in a manner similar to the classical formalization of failure detectors [22]. The two algorithms they propose

accumulate information about broadcast propagation paths over epochs in order to construct local reachability information. When the set of nodes in the system is known beforehand [25], a partition is detected when some of these nodes become unreachable (possibly because of crashes). This approach is extended to systems with an arbitrary number of unknown participants [2], for which the detector is able to return the set of nodes present in the local partition, provided the local partition eventually stabilizes. The accumulation of network participants in a list is similar to the way we accumulate members in our filters. For large networks, however, such an explicit approach is likely not to scale, contrary to the strategy we advocate.

Ritter *et al.* [67] propose an approach to detect partitions in MANETs in which a subset of active nodes exchange beacon messages that traverse the network. The proposed heuristic tends to position active nodes at the border of the network, in order to maximize the network nodes covered by a beacon propagation path. When beacons repeatedly fail to propagate between two active nodes, a partition is suspected. In contrast to the approach we propose, this strategy assumes that border nodes can be reliably detected, and only change slowly, which might not be the case.

In [48], Khelil *et al.* present a broadcast strategy for partitionable MANETS based on hypergossiping, the selective re-broadcasting of partially broadcast messages. This strategy includes a mechanism to detect partition joins, i.e. the rejoining of the two parts of a previously disconnected network. This mechanism exploits Last Broadcast Received (LBR) lists, a list of the IDs of the k last broadcast messages received by a node. Nodes periodically exchange this list, and conclude that they are rejoining a partitioned subnetwork when their local LBR substantially differ from that of their neighbors. Because the main goal of this approach is to maximize the delivery ratio of system-wide broadcasts, the partition join detection mechanism tends to err on the side of over-detection, with numerous wrong detection decisions in some instances [49].

Cut detection Cut detection is a problem related, but distinct from partition detection in MANETs, and focuses on (mostly static) Wireless Sensor Networks (WSN), in which sensors forward their readings to dedicated sink nodes. A cut occurs when some sensor nodes become disconnected from the sink.

The work of Barooah *et al.* [7] allows each sensor node to detect if it becomes disconnected from the sink, and if it remains connected, to detect whether other sensor nodes have become disconnected. The work in [82] considers only the second problem and adds consideration about energy and robustness to malicious nodes. Because of the specific topology of sink-based sensor networks, these approaches are however not applicable to our scenario.

Partition prediction Some works try to predict partitions before they happen, but require more powerful primitives than our proposal. Some papers [58] for instance use GPS information (regarding both location and speed) to build a mobility model of the

network and predict when nodes are likely to get out of range. Other proposals such as [39] assume the existence of a distributed algorithm that returns the set of disjoint paths between two nodes, and predict partitions based on the number of paths and their length. By comparison, our solution makes no assumption regarding the higher level capabilities of a network, and only assumes a one-hop broadcast primitive.

2.3 Conclusion

In Table 2.1, we summarize the various technologies and approaches we described above. A ✓ means the corresponding approach realizes a given property, a ✗ means the opposite. We indicated "~" when the property is partially realized or can be realized or not depending on the context and the other technologies involved in a real use case. Finally, we indicated "N/A" when the property being realized or not does not depend on the approach considered.

There are a few interesting patterns to pick up: The majority of approaches, especially the more feature-rich like MESOS or Kubernetes, are centralized, even those targeted at distributed system from the start like Kairos. Higher-level abstractions and ease of programming, such as macro-programming and modularity are generally coupled with elements of a centralized management that ensures the interface between the high-level concepts and the low-level deployment. This usually entails some non-optimal performances regarding the more technical properties like scalability or resilience. On the other end of the spectrum, gossip protocols have great technical properties (resilience, scalability, etc.) but do not offer programming facilities to manipulate higher concepts and are mostly developed from the behavior of individual nodes.

In conclusion, in this chapter, we presented the current approaches in terms of programming frameworks and adaptation mechanisms, and how they are currently realized in distributed systems, and we identified in the technological landscape a lack of high-level, easy to program approaches which can also work under harsh conditions and scale properly while being fully decentralized, necessary to realize our vision of opportunistic self-adaptive systems managed with a holistic approach

In the next chapter, we focus on the first face of this vision: a holistic approach that consider distributed systems as a whole entity, without having recourse to centralization or sacrificing performances.

Properties \ Approaches	Frameworks for WSN	Acquisitional Query Processor	Tuple Spaces	Neighborhood Primitives	Fragmented Objects	Epidemic Protocols	Cluster Management	Containers Management
Examples	Kairos, Regiment	TinyDB, Cougar, MauveDB	TineeLime, TOTA	Hood, Abstract Regions, Logical Neighborhoods	FORMI	TMan, Polystyrene	MESOS	Kubernetes
Macro-programming	✓	✓	×	×	~	×	×	×
Modularity	×	×	×	✓	✓	×	×	×
Node placement	×	×	N/A	N/A	×	N/A	~	~
Node-independent	~	✓	✓	N/A	✓	×	✓	✓
High-level coordination	~	✓	✓	~	N/A	×	✓	✓
Scoping	N/A	N/A	×	✓	×	×	✓	×
Scalability / Elasticity	N/A	✓	×	~	✓	✓	✓	✓
Decentralized	×	×	~	~	✓	✓	×	×
Resilience	N/A	✓	N/A	✓	✓	✓	✓	~

Table 2.1: Summary of various approaches and their respective properties

Chapter 3

Hollistic construction: PLEIADES

In Chapter 1, we argued for a two-fold vision: on one hand, a holistic approach that considers a system as a whole and moves away from the behavior of individual components; on the other hand, opportunistic systems with smarter basic blocks. In this chapter, we explore the first face of this vision, and we demonstrate how such a holistic approach could be realized, combining a high-level description by assembly with a stack of concurrent and collaborating self-organizing overlays. We focus on a specific problem: the development, deployment and maintenance of complex logical overlays that realize elaborate topological structures. We consider this challenge in the context of a flat network, where every node can send messages to any other node and where peer-sampling is easily available.

3.1 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 [40]. 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 [51, 34, 60, 73]. For instance, *MongoDB* [60], a popular document-oriented no-sql databases, uses a star topology between sets of nodes organized in cliques (Figure 3.1a). Similarly the cross-datacenter replication feature of *Riak* [73], a production-level key-value datastore, relies on the connection of multiple rings across geo-distributed datacenters (Figure 3.1b). These services are often fur-

(a) MongoDB (b) Riak

Figure 3.1: Some complex topologies encountered in modern distributed systems

ther embedded within micro-service architectures [74] resulting in increasingly complex distributed topologies, that can be hard to describe, monitor, and adapt.

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 [75, 59], or component frameworks [27, 20, 71]), 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* [11], from which whole distributed systems can be *incrementally assembled*. Furthermore, 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 Pleiades , an assembly-based programming framework for the implementation of complex distributed topologies. Pleiades provides developers with a high level component-based programming model [27, 20], and exploits self-organizing overlays [79, 10, 42] to map at runtime a developer's high-level description of a complex distributed topology onto a concrete infrastructure. Pleiades 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.

Pleiades goes beyond traditional component-based framework for distributed sys-

tems 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.

Indeed, due to the size and complexity of these systems, and the unpredictability of the environments, this composability cannot be a rigid construct, but must instead, we argue, go hand-in-hand with advanced *self-organization* capabilities.

Current self-organizing overlays [42, 10, ?] exploit epidemic (or gossip) interactions to progressively organize nodes along a predefined topology —from a random network [43] to a ring or torus [78, 42] to an hypercube. These topologies can be used to support the many P2P- and cloud-based applications that have been proposed for over a decade now, such as VoIP (e.g. Skype), streaming [84], pub-sub [23], and storage [62, 34].

Typically, gossip overlays assume that node profiles can be sorted according to a ranking function which is used uniformly across the system, and that finding the *"right" neighbor* is just optimizing this ranking function over their neighborhood. However, more and more applications require much more complex topologies that can be hard to obtain *via* the traditional protocols [34, 51], because, for more convoluted topologies such as a ring of cliques, such a ranking function can be hard to express.

Our proposal goes beyond this limitation and relies on the following key points:

- 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 PLEIADES, a component-based framework that implements our programming model. PLEIADES 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 PLEIADES and perform a thorough evaluation that demonstrates its genericity, expressiveness, low-overhead, and adaptability.

The remainder of this chapter is structured as follows. Section 3.2 presents the context and challenges motivating PLEIADES; Section 3.3 introduces our component-based programming model, and the framework built upon it, and explains how PLEIADES hides the intricacies of our approach from the programmer; Section 3.4 evaluates the scalability and efficiency of our framework with a proof-of-concept implementation; and Section 3.5 concludes.

3.2 Problem, Vision, & Background

3.2.1 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* [34] 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 [60]. *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 [73]. 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 [74, 57].

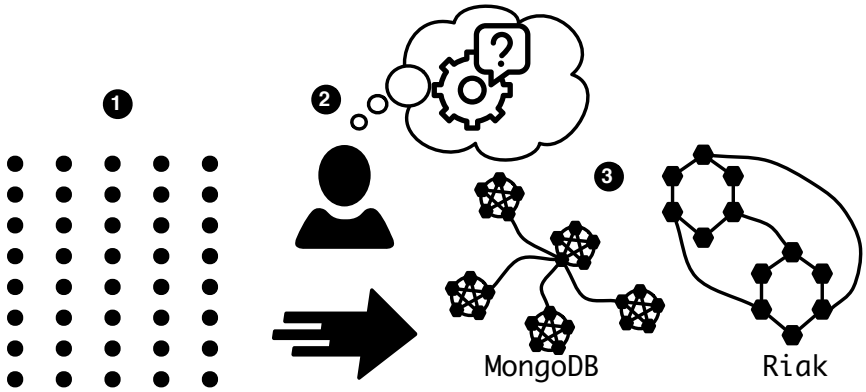


Figure 3.2: Creation of complex systems of systems

Provisioning, deploying and maintaining these distributed topologies is unfortunately a cumbersome and error-prone task. Developers must provision nodes (Figure 3.2 ❶), specify the local services they should execute (Figure 3.2 ❷), and connect these local services into an appropriate topology (Figure 3.2 ❸), according to the service and application's needs. Many of these tasks can be alleviated using deployment automation tools such as *Borg* [77], *Kubernetes* [21], *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 chapter, 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 PLEIADES, an assembly-based topology programming framework that free developers from low-level topology deployment and maintenance. PLEIADES 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.

3.2.2 Key challenges and roadmap

Explicitly building the targeted network topology (Figure 3.2 ❸), without a principled and systematic programming model, is a tiresome and cumbersome task for programmers. A possible and promising approach is to build such complex topologies as an assemblage of simpler shape that are then connected altogether. To reach this aim, we introduce PLEIADES, an assembly-based topology programming framework that harnesses the autonomous properties of self-organizing overlays To deliver this model, PLEIADES must overcome dynamically the following key challenges:

- provide a high-level description of the target composite topology;
- map "system-level" components to nodes;
- realize the dynamic bindings that connect individual shapes according to the developer's high-level plan;
- maintain and handle communications among different components.

3.3 The PLEIADES framework: DSL, library & runtime

The PLEIADES framework comprises (i) a DSL, (ii) a basic shape library, and (iii) a runtime. The PLEIADES DSL is simple and expressive enough to describe a large array of topologies that can be difficult to achieve with earlier methods. The PLEIADES DSL achieves this goal by allowing developers to construct a complex topology by assembling simpler blocks, termed *shapes*. To support this process, the PLEIADES framework provides a shape library that includes, by default, basic shapes that implement simple topological constructs such as rings, grids, etc. Finally, the PLEIADES

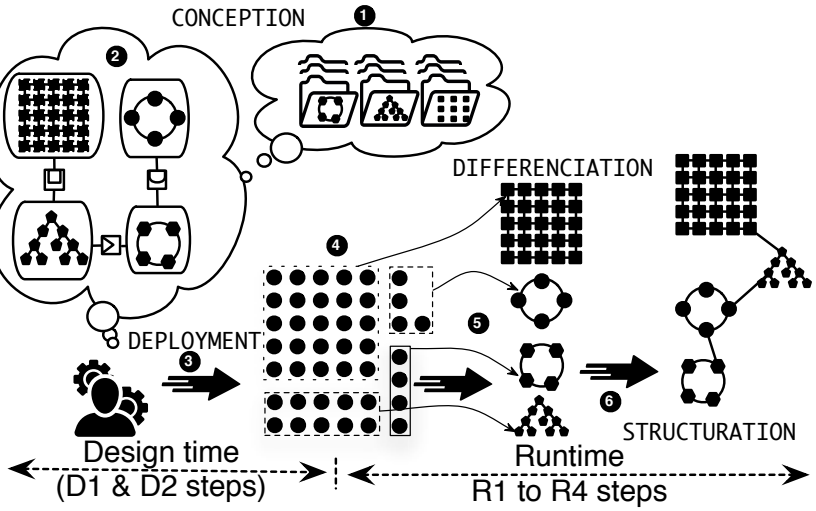


Figure 3.3: Pleiades overall approach

framework comes with a runtime that handles under-the-hood the role allocation and the differentiation of nodes that belong to different shapes.

In the following, we describe the various part of the Pleiades framework in details. Subsection 3.3.1 starts with a general description of the framework architecture and organisation. Subsection 3.3.2 then explains how shape templates are defined in the Pleiades library, and how ports are added to connect shapes. Subsection 3.3.3 presents Pleiades DSL, and illustrates its use to describe complex target topologies. And finally, Subsection 3.3.4 and 3.3.5 discuss the various protocols comprising Pleiades run-time.

3.3.1 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 3.4. 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 [46], 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 3.4, 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

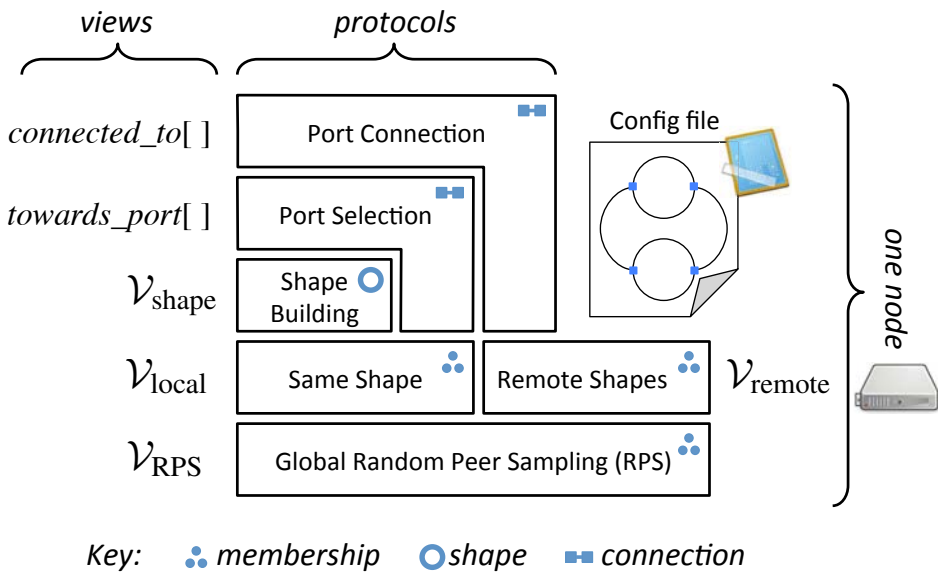


Figure 3.4: PLEIADES consists of 6 self-stabilizing protocols that build upon one another to enforce the structural invariant described in a configuration file distributed to all nodes in the system.

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, and is typically where one would plug the shape library to easily get basic 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 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 3.4 use the view constructed by lower protocols to construct their own view.

3.3.2 Shape templates and port definition

In PLEIADES, a shape s is a subset $N_s \subseteq N$ of message-passing 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 3.4 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:

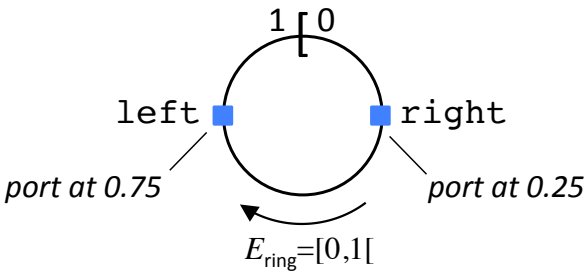


Figure 3.5: A simple ring template can be defined using $E_{\text{ring}} = [0, 1[$ as position space, a random projection function, the modulo distance, and a shape fanout $k_{\text{ring}} = 2$.

- the definition of a *position space* E_{tplate};
- a *projection function* $f_{\text{tplate}} : N_s \mapsto E_{\text{tplate}}$ that assigns a position in E_{tplate} to each node selected to be part of an instance of tplate;
- a *ranking function*[1] $d_{\text{tplate}} : E_{\text{tplate}} \times E_{\text{tplate}} \mapsto \mathbb{R}$;
- a *number of neighbors* (or shape fanout) per node, k_{tplate}.

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

For instance, a naive version of a self-stabilizing ring can be defined as follows (Figure 3.5):

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

This setting places nodes from N_s 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 $k_{\text{tplate}}/2$ predecessors and $k_{\text{tplate}}/2$ successors as neighbors of each node, to prevent clustering. See [42, 62, 72].)

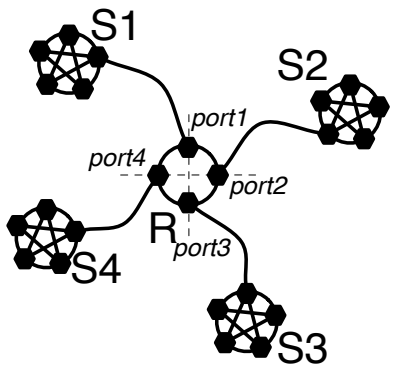
These template definition may look tedious, but they only need to be done once and for all. Ideally, PLEIADES would provide a ready-to-use *shape library* that contains a set of such templates implementing various common elementary topologies (ring, tree, torus, grid, etc.), that a developer can combine to build a complex distributed topology.

In addition to the internal structure described by its template, a shape instance also needs to define a set of *ports* to which other shape instances 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 3.5, we may define two ports, named `left` and `right`, by associating them with the positions 0.25 and 0.75 within the identifier space $E_{ring} = [0, 1[$.

[1] As mentioned in [42], self-organizing overlays employ ranking functions that cannot always be defined as global *distance* functions.

3.3.3 PLEIADES DSL and configuration file

The PLEIADES DSL enables programmers to describe their expected complex distributed topology, i.e the *target topology*, by specifying the shape templates they want to instantiate, the ports for each instance, and how to connect them together. For instance, the code given in Figure 3.6 describes how to create the topology displayed on the side. The described topology is nearly similar to what is actually used in current applications such as MongoDB to create sharded databases. The only difference is that we used a ring instead of a clique for the Router instance, solely to demonstrate how easy it is to use different templates in the same target topology (e.g. 1 Ring instance and 4 Cliques instances here).




```
1  import { Clique, Ring } from defaultShapes
2
3  instance 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 instance Shard1 as s1 using Clique
11 s1.nodes(random,0,2)
12 s1.addPort(leader,0.254)
13
14 instance Shard2 as s2 using Clique
15 s2.nodes(random,0,2)
16 s2.addPort(leader,0.943)
17
18 instance Shard3 as s3 using Clique
19 s3.nodes(random,0,2)
20 s3.addPort(leader,0.159)
21
22 instance Shard4 as s4 using Clique
23 s4.nodes(random,0,2)
24 s4.addPort(leader,0.681)
25
26
27 connections {
28         (r.port1, s1.leader),
29         (r.port2, s2.leader),
30         (r.port3, s3.leader),
31         (r.port4, s4.leader)
32 }
```

Figure 3.6: MongoDB-like topology: a Star of Cliques connected through a Ring

To create this target topology, i.e. a star of cliques connected through a ring, a developer has to go through the following process:

1. First, elementary topologies need to be imported. So both a ring and a clique template are imported (line 1) from the shape library.

2. Next, each template is instantiated a number of times. On one hand, an instance

named Router that implements the Ring template (line 3). On the other hand, four instances named Shard 1 through 4 that implement the Clique template (line 10,14,18,22). The construct `instance ... using ...` enables the creation of a new instance based on an existing template that has been previously imported.

3. As a shape instance is a subset of nodes, the next required step is to specify how the available nodes of our system will be allocated at runtime to each defined instance. This is done using the `nodes(<strategie>,...)` construct, with parameters for the specific strategy chosen filling the blank. For example, assuming all nodes are identical and locally have access to a random number generator, we can perform, at runtime, a uniform distribution of the available nodes among the five components, by calling the strategy `random` with parameter 0.2 for each component (line 4, 11, 15, 19, 23).

4. Further, to connect instances together, at least one port needs to be defined for each instance. Adding a port to an instance at a specific position in its position space E_{tplate} is performed *via* the construct `addPort(label, position)`. As the Router instance has a Ring topology, a port is created at each of the four positions 0, 0.25, 0.5, 0.75 within the position space $E_{ring} = [0, 1[$ (See line 5-8). For the four other instances, a port is setup at an arbitrary position in the position space E_{clique} of each component and the node in charge of that port will be considered the leader of that clique (line 11,15,19,23). Once created, a port can be directly addressed by its *label* from the corresponding component.

5. Finally, to assemble the various shape instances and build the target topology, a list of connections between the aforementioned ports is created *via* the construct `connections { (port1,port2), ...}` (line 27-32).

There are a couple important details to call out:

Local knowledge only: For node placement, the main constraint here is that a node which joined the system and just received the configuration file needs to interpret it and, using only its local knowledge of the system, determine in which shape instance it must participate. Consequently, strategies can use any *local property*, or *criterion*, that a node is able to self-evaluate: available bandwidth, CPU power, storage space, local fan-out, etc. In a cloud system, where nodes represent virtual machines in servers and are all functionally identical, the allocation can be done at random, as illustrated just above. However, in more heterogeneous systems with specialized devices such as a Smart Building, each device can be allocated to a fitting instance based on its capabilities and the instance needs.

Simple but powerful allocation model: Despite being restricted to local knowledge only, this allocation mechanism can exploit a deceptively wide range of strategies,

depending on uses cases : nodes in a particular *location* may be constrained to only join certain shapes *instances*, or nodes with certain *properties* may be forbidden to join certain shape *templates*, and so on. The main limit of this simple but powerful model is that it cannot express constraints that restrict multiple nodes simultaneously, such as: "I want one node from each of three datacenters in that instance."

Port selection is automatically resilient: In the port definitions, note that each Clique instance will (eventually) reach a consensus on which node is the leader because the configuration file is the same for every single node. The leader was, in fact, selected when writing the configuration file and this is not a real election process. However, and this is one of the most attractive aspect of PLEIADES, since the configuration file does not specify an individual node but only a position, the system will automatically adapt at run-time if the chosen leader crashes or moves away from the position specified in the configuration. More details on this automatic resilience below.

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

- **(D1)** use the PLEIADES DSL to describe which shape templates should be instantiated, how they should be connected to each other via ports, and provide node-provisioning policies to determine how individual node should be allocated to components (See Figure 3.3 ❶ ❷);
- **(D2)** compile and deploy the resulting PLEIADES configuration file to a set of nodes executing the PLEIADES runtime (Figure 3.3 ❸).

On receiving this file the runtime will

- **(R1)** allocate each individual node to a shape instance (determining N_s for each shape N_s): by default each node belongs to one and only one shape instance, but it would be relatively easy to extend our model to allow further flexibility; (Figure 3.3 ❹)
- **(R2)** create each shape instance's internal topology by executing the Shape Building protocol according to the corresponding template;
- **(R3)** identify within each instance which node(s) should manage this instance's individual ports ; (Figure 3.3 ❺)
- **(R4)** finally, connect the resulting ports according to the configuration file's specifications. (Figure 3.3 ❻)

These runtime steps occur in a fully decentralized manner, without resorting to any centralized entities, a key property for the scalability and resilience of our approach. In the following subsections (3.3.4 and 3.3.5), we present how the runtime fulfills its missions (Steps **R1** through **R4**).

Table 3.1: 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 3.2: 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

3.3.4 The Membership and Shape Building protocols

The first step for a node joining the system is to get the configuration file, interpret it, and determines to which shape instance it must participate. At that point, just after joining an instance, a node possesses no information about which other nodes belong to the same instance, or how to contact other nodes in other instances. This information is provided by PLEIADES's three membership protocols. The Global *Random Peer-Sampling* (RPS) protocol [43] 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 instance it resides on, and its position in this instance.

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 Subection 3.3.5), to create and maintain the connections between the shape instances 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 3.1. 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.

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.* [43]. RPS protocols converge towards a constantly changing overlay that is close to a fixed-degree random graph. This graph shows a short diameter, which is

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

▷`Remote request to` q

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

▷`Truncation`

6 $n.\mathcal{V}_{\text{local}} \leftarrow$ up to s random nodes $\in cand$

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.

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 6).

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 4.4) which allows the system to converge and reach a stable state quickly and efficiently.

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

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

▷Keep one node per ''close'' shape

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

6 $\quad cand_{\mathsf{s}'} \leftarrow \{n' \in cand \mid n'.shape.id = \mathsf{s}'\}$

7 $\quad$ **if** $cand_{\mathsf{s}'} \neq \varnothing$ **then**

8 $\quad\quad n.\mathcal{V}_{\text{remote}}[\mathsf{s}'] \leftarrow$ 1 random node $\in cand_{\mathsf{s}'}$

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 6 to 8 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 6 filters candidate nodes from shape s' into $cand_{\mathsf{s}'}$, and lines 7-8 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').

Shape Building Protocol

We use a variant of *Vicinity* [79] to organize the nodes that have joined a shape s into the basic topology prescribed by the shape's template tplate. 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

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 ∈ n.shape.ports do
      ▷Find closest node to port k among local nodes
2     cand ← n.V_local ∪ n.V_shape ∪ {n, n.towards_port[k]}
3     closest ← GETCLOSEST(cand, k, n.shape.template)
4     n.is_port[k] ← (n = closest)
5     if n.is_port[k] then
6         n.towards_port[k] ← n
7     else
          ▷If n is not port node, remote request to closest
8         n.towards_port[k] ← closest.towards_port[k]
```

Function getClosest($cand, k,$ tplate)

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

```
1 closest ← arg min_{p∈cand} (d_tplate(p.id, k.id))
2 return closest
```

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.

3.3.5 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 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 3.2.

GETCLOSEST($cand$,k, 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 3.3.2), 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.

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 $n.connected\_to[k_1] \leftarrow closest.towards\_port[k_2]$

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.

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

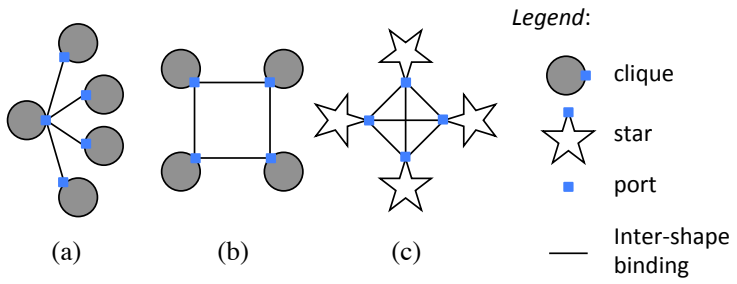
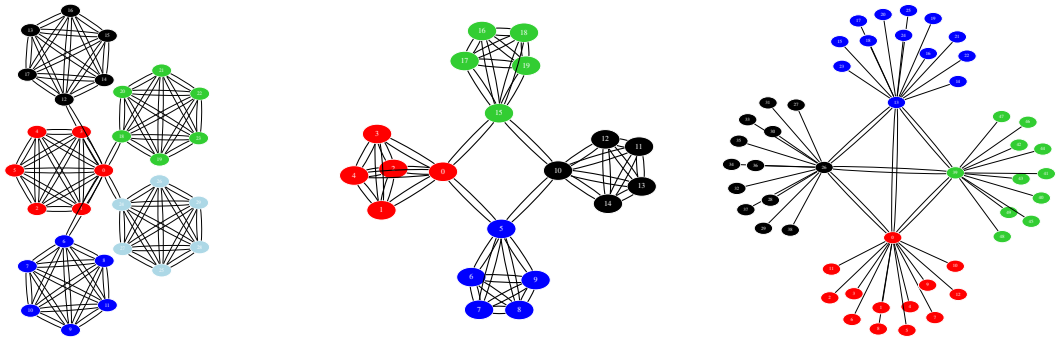


Figure 3.7: A graphical representation of the PLEIADES configuration files used to create the systems shown in Figure 3.8.



(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 3.8: The resulting topologies corresponding to the configurations of Figure 3.7 (after 10 rounds of simulation).

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.

3.4 Evaluation

In this section, we first discuss our evaluation set-up (Section 3.4.1) before briefly illustrating how PLEIADES can be used to create a range of advanced distributed structures (Section 3.4.2). We then evaluate the performance of PLEIADES without

reconfiguration or failures, in terms of convergence speed, scalability, and communication overhead (Section 3.4.3). 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 3.4.4).

3.4.1 Evaluation set-up and methodology

We implemented the protocols that make up PLEIADES on top of PeerSim [61], 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 [79]. 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.

3.4.2 Examples

Figure 3.7 graphically presents three configuration files used by PLEIADES to construct the three distributed systems shown in Figure 3.8. We used graphical representations for the sake of clarity, but the real files are similar to the one already shown in Figure 3.6. These three examples connect simpler shapes together (*cliques* and *stars*, shown symbolically in Figure 3.7 and with different colors in Figure 3.8). The resulting toplogies can be found in real-world applications, such as database sharding (Figure 3.8a), distributed key value stores (Figure 3.8b) or partially decentralized services using super-peers (Figure 3.8c).

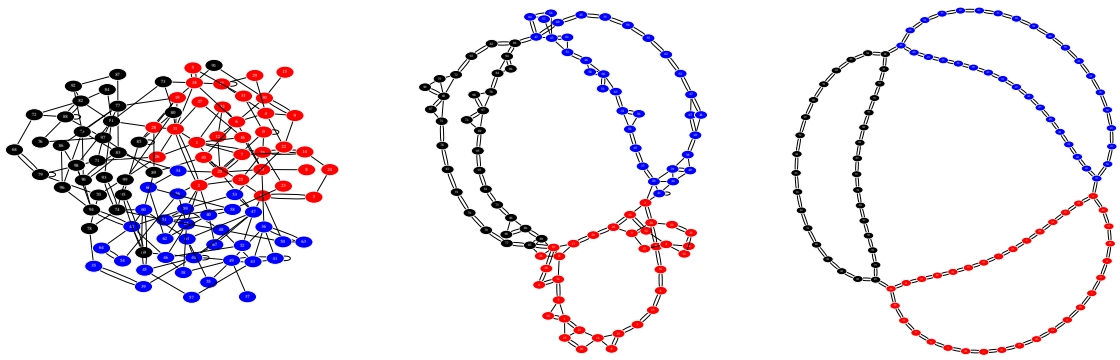
These three examples illustrate PLEIADES's simplicity of use and expressiveness: a few basic shapes suffice to create an infinite number of variations that can be tailored to an application's needs.

3.4.3 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 3.4.3), *scalability* (Section 3.4.3), and *communication overhead* (Section 3.4.3).

Convergence

We evaluate PLEIADES's convergence on a scenario comprising three rings connected into *a ring of rings*, whose configuration is represented in Figure 3.10. Figure 3.9 shows



(a) Random initial state

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

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

Figure 3.9: A system of 100 nodes converges in 6 rounds towards three connected rings (colored in blue, red, and black).

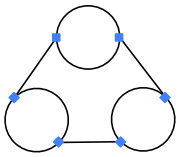


Figure 3.10: The PLEIADES configuration used in Figure 3.9.

the execution of PLEIADES with this configuration on 100 nodes at three stages of the execution: after initialization (Fig. 3.9a), while the system is converging (Fig. 3.9b) and once converged (Fig. 3.9c). 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 5 seconds (a realistic assumption in light of PLEIADES's low communication costs as we will see in Section 3.4.3), 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 3.11 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 [42, 79]. The sequence of convergence roughly follows the dependencies between the protocols illustrated in Figure 3.4: 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

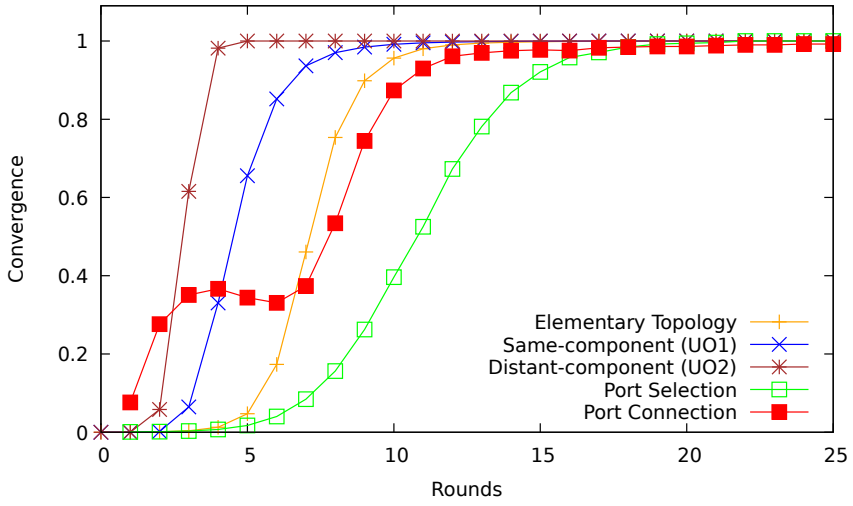
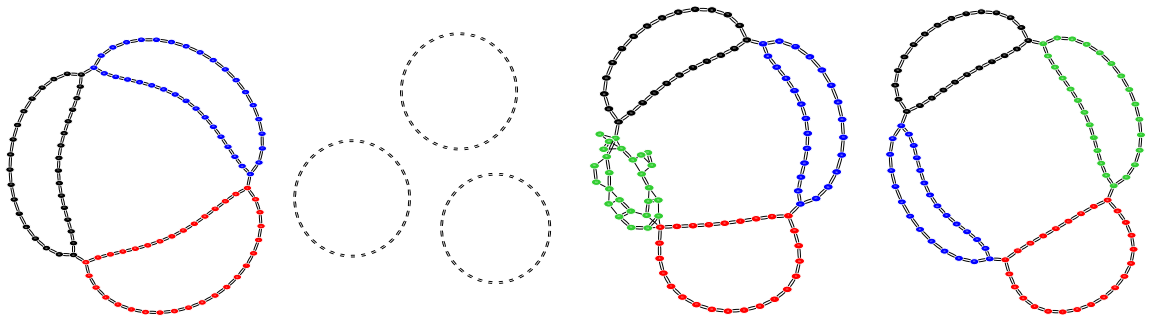


Figure 3.11: Progress of the different protocols of PLEIADES over time (in rounds) for a ring of rings with 25,600 nodes and 10 rings. Except for Port Connection, all protocols experience a rapid phase change.

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%.



(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 3.12: Dynamic reconfiguration and convergence to a new stable state.

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

In Figure 3.13, 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 3.14, 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.

Communication overhead

Compared to an ad-hoc approach optimized for a given problem, PLEIADES 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

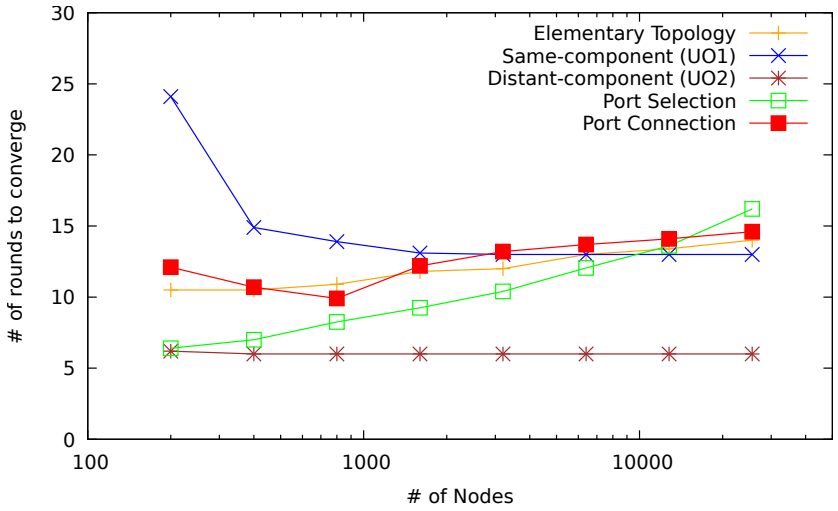


Figure 3.13: Convergence time of the PLEIADES protocols for a system of 20 connected rings (a *ring of rings*), for various system sizes. PLEIADES converges rapidly and scales well with the number of nodes.

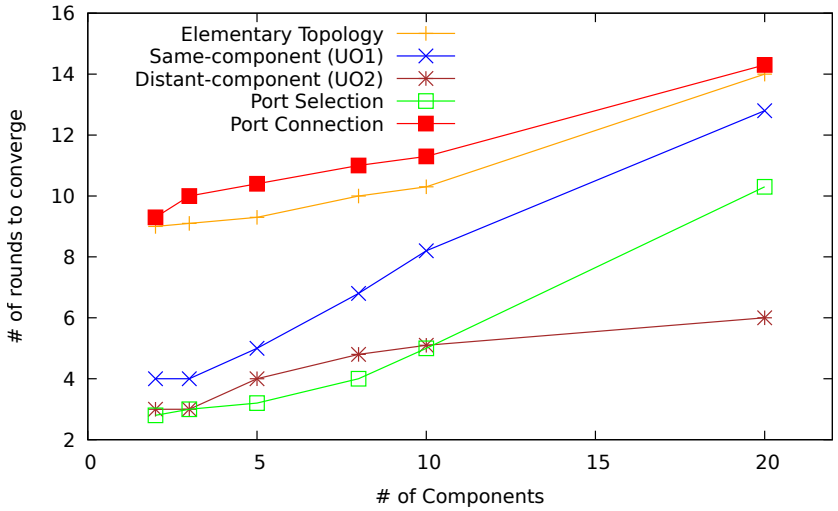


Figure 3.14: Convergence time of the PLEIADES protocols for a system of 25,600 nodes implementing a *ring of rings*, for various numbers of rings. The convergence time of PLEIADES only slowly increases with the number of individual rings.

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 3.15 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

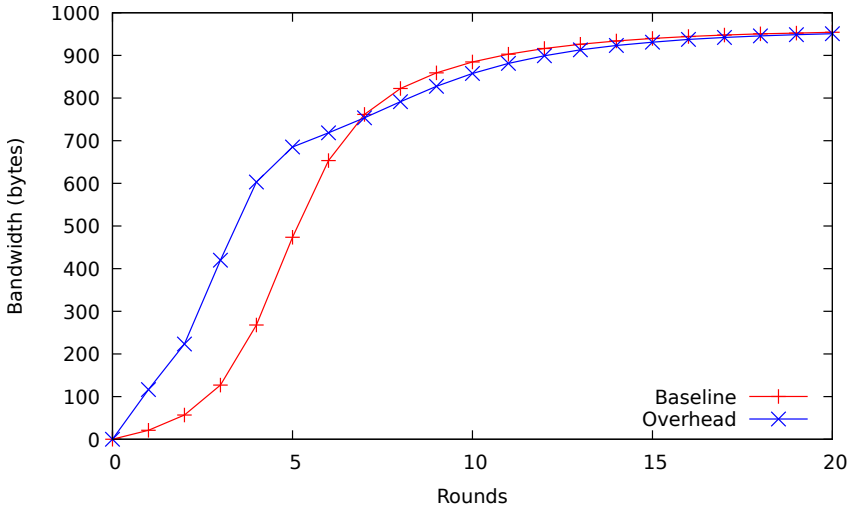


Figure 3.15: Bandwidth overhead of PLEIADES over the shape building protocol, per node, per round (20 shapes, 25,600 nodes). Both protocols peak once all views have stabilized, and remain below 1kB (2kB in total).

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 3.18 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 3.18 for 50 shapes, the bandwidth ratio is around 2, which in absolute value represents 1900 bytes, so it represents a very negligible amount.

3.4.4 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 3.4.4); secondly, an on-the-fly reconfiguration of the target topology, changing the number of basic shapes in the system (paragraph 3.4.4).

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 3.17.

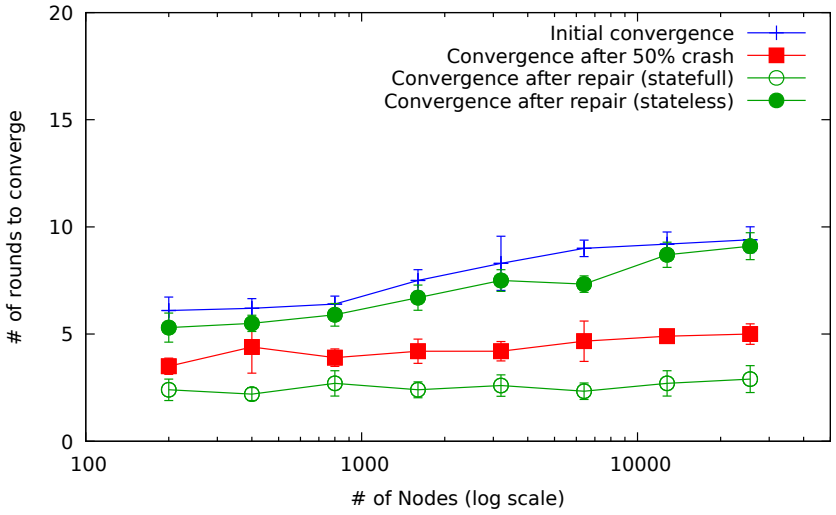
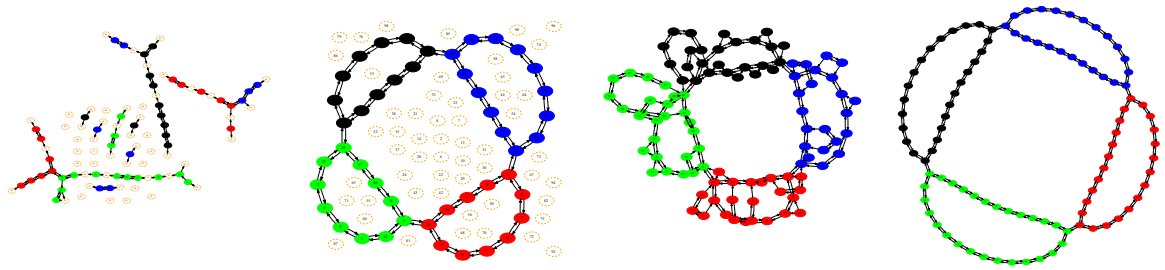


Figure 3.16: PLEIADES's convergence time after half of the nodes have crashed, and after re-injecting new nodes (4 connected rings, note the log x axis). PLEIADES's stabilization speed is logarithmic in the system's size.



(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 3.17: Resilience and self-repair after a dramatic crash or a large node injection.

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 (3.17a), and we let the system converge towards the new resulting target topology (3.17b). Finally, we simultaneously inject as many nodes as crashed earlier (3.17c) and we let the system converge back to the original target topology (3.17d). 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 3.4.3 are

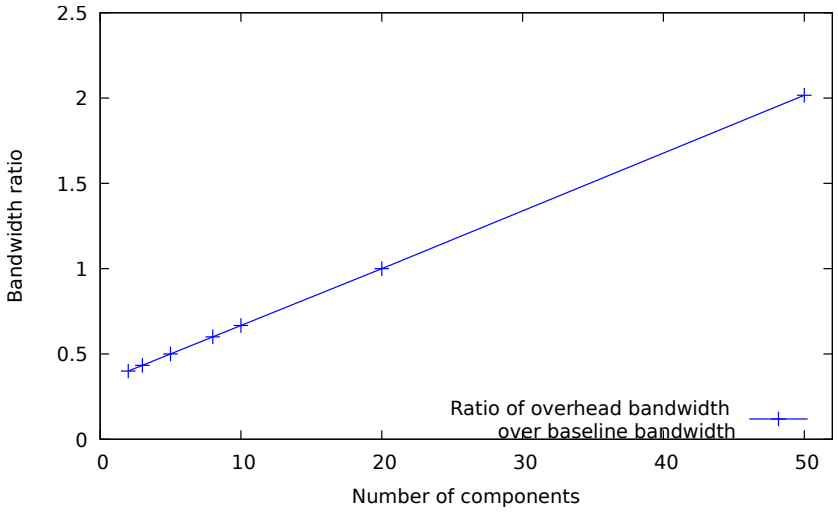


Figure 3.18: Evolution of the bandwidth overhead of PLEIADES (ratio) vs. the number of basic shapes (25,600 nodes, stable state). PLEIADES's overhead remains very small even for 50 basic shapes (< 2kB in absolute value).

satisfied. Figure 3.16 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.

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 each node simply randomly decides to migrate to the new shape with probability 1/4, resulting in 4 new shapes each with three quarters of the nodes in the original shapes. Many other policies may be envisioned, from assigning nodes without taking their prior assignment into account at all, to adding new nodes into the system specifically to the new shape and preserving the existing shapes (only changing the links between shapes), or even splitting one of the existing shapes in two and leaving the other two shapes undisturbed.

The "right" policy to use is obviously context-dependent, based on what function is actually run on top of the topology, and so how to choose a reconfiguration policy is out of the scope of this chapter. The simple policy we used, though, is one of the possibilities that create the largest amount of disturbances in the system, and showing that PLEIADES is able to handle it efficiently is a convincing argument that it is able to handle most reasonable and useful policies in practice. Note however, that just as the initial assignment, the reconfiguration policy can only use local knowledge.

At a given round (Figure 3.12b), the new configuration is sent to all the nodes, and some of them are allocated to the new shape. Only 2 rounds later (Figure 3.12c), 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 3.12d). All measurements presented in Section 3.4.3 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.

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.

3.5 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 proposed the PLEIADES framework. PLEIADES is based, on one hand, on software engineering design principles such as encapsulation and programming by assembly to rise the level of abstraction, and on the other hand, on self-organizing overlays to handle the low level work automatically and mask the growing complexities. However, PLEIADES goes one step further 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 PLEIADES framework comprises three core elements: (i) a shape library, (ii) a DSL, and (iii) a runtime. The PLEIADES shape library provides a set of shape templates for elementary topological construct that developers can pick up and combine easily. The PLEIADES DSL enables developers to write a PLEIADES

configuration file that describes a complex distributed topology in an easy manner: mainly by specifying the shape templates they want to instantiate and how they wish to connect them together. Finally, once the PLEIADES configuration is ready and deployed, the runtime on each node combines six self-organizing protocols that work together to construct and maintain the target topology prescribed in the configuration.

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 using very limited bandwidth. PLEIADES further scales logarithmically in the number of system's nodes, and close to linearly in the number of elementary shapes.

Finally, we have demonstrated that our approach enables to create in an efficient and scalable way a wide range 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 PLEIADES: (i) are able to converge quickly 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.

PLEIADES thus constitutes convincing evidence that the vision we proposed in Chapter 1 is indeed a realistic solution to the challenges faced by modern distributed systems. Holistic approaches are viable, and accompanied by the proper infrastructure and low level automation, they enable developers to tackle and manipulate the increasingly complex modern systems.

In the next chapter, we will now move onto the second face of our vision: make the basic building blocks of distributed systems self-adaptive and able to react to changing circumstances, to combine and compose opportunistic systems.

Chapter 4

Detecting environmental changes: Mind-the-Gap

In Chapter 1, we argued for a two-fold vision: on one hand, a holistic approach that considers a system as a whole and moves away from the behavior of individual components; on the other hand, opportunistic systems with smarter basic blocks, able to self-organize, react and adapt to changing circumstances, to automate the low-level behaviors masked by the aforementioned approach. In the previous chapter, we demonstrated how such a holistic approach could be realized, combining a high-level description by assembly with a stack of concurrent and collaborating self-organizing overlays. In this chapter, we now move onto the question of making self-adapting and opportunistic systems.

More specifically, we focus on the first step of the self-adaptation process: the monitoring of the system's environment and the detection of changes in circumstances. Additionally we address this question in one of the most difficult contexts, Mobile Ad-hoc Networks: they are highly dynamic systems, with usually very limited capabilities in terms of battery life, communication range, processing power, and so on. Finally, we target a specific class of events: partitions and other large changes in network connectivity.

4.1 Introduction

Mobile Ad-hoc Networks (MANETs) are decentralized wireless systems composed of physically- mobile nodes. They do not rely on any fixed network infrastructure, but instead exploit mesh networking protocols [1, 69] to overcome the mobility of nodes and the imperfect and unpredictable nature of wireless communication, leading to collisions and messages loss. Because they operate in open, dynamic and sometimes hostile environments, MANETs can easily become *partitioned*, i.e. some parts of the network are unable to reach some other parts, for instance because some key nodes have failed, or because nodes have moved out of reach from one another. When this

occurs, the network becomes disconnected, and most routing mechanisms cease to function[1].

A partition can be mitigated by deploying additional resources such as a supporting Flying Ad-Hoc Network (FANET) [8, 38], or through opportunistic composition with third party systems [11]. Before any such mitigating action can be taken, however, the partition must first be *detected*. Detecting a partition in a fully decentralized and autonomous system such as a MANET is a challenging task: MANETs typically lack any central element, forcing each node to build its own perception of the network's current state, and do so quickly enough so that the dynamically evolving network that is a MANET has not changed too much. This decentralized monitoring must also remain extremely lightweight in order to meet the memory, CPU, and energy constraints of mobile nodes.

Previous proposals to detect partitions in MANETs have either assumed extended node capabilities [39, 58] (such as a GPS sensors or accelerators), thus limiting their applicability to high-end deployments, or have attempted to construct explicit membership and path information [2, 25, 67]. As MANETs reach several hundreds of nodes, gathering explicit node lists is increasingly problematic: explicit representations incur important communication costs and lead to a rapid depletion of energy resources.

In this work, we tackle partition detection in MANETs by coupling a *probabilistic compact representation* of a network's composition with a *periodic aggregation procedure* inspired by gossip protocols [41, 45]. These two primitives in tandem allow us to arbitrate the inherent trade-offs arising between speed, accuracy, and cost of the detection (in terms of communication overhead), and thus offer an adaptable range of guarantees tailored to each system's requirements in a lightweight, decentralized and accurate fashion. We instantiate them so that partitions can be self-detected by a MANET by identifying temporal discrepancies, or detected by a second network monitoring spatial discrepancies. The choice between these two use cases depends on whether mitigating measures should be triggered by the partitioned network (such as switching to an alternative wireless technology [35]), or by some external system (such as offering bridging capabilities by a FANET [8, 11]). The protocol we propose relies on the following key points:

1. We use random bit signatures to concisely encode a MANET's set of nodes, or *membership list*, and show that this compact and probabilistic representation can detect large connectivity changes with very high probability.

2. We present partition-detection algorithms that combine our probabilistic representation with a periodic aggregation procedure, and offer both an internal self-detection mechanism and an external third-party detection service.

[1]Delay Tolerant Networking (DTN) protocols [30] are a noticeable exception (albeit with strong operational and applicative constraints), which we do not consider here.

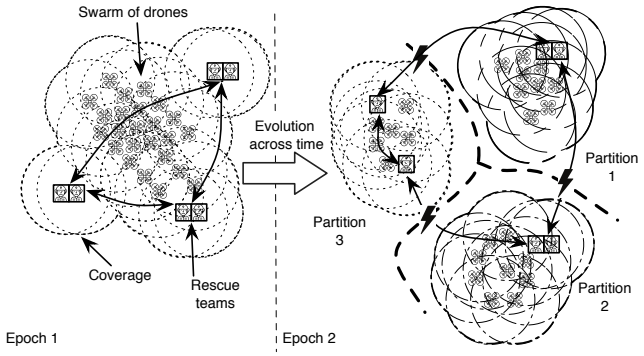


Figure 4.1

3. We develop a theoretical analysis to tune implementation parameters based on the environment in which the network is deployed. We show that as long as the filters are not completely filled with '1's , we detect close to 100% of partitions and almost none of the non-partition events.

4. We demonstrate the practical relevance of our approach through an extensive series of simulations. As an example, we show that even with 40% message loss, performances are still satisfying with an error rate of detection below 10%.

The remainder of this chapter is organized as follows. We present our approach in Section 4.2, analyze it formally in Section 4.3, and present an in depth experimental evaluation in Section 4.4. Finally we conclude in Section 4.5.

4.2 Approach

4.2.1 Overview

A simple approach to detecting a partition, or a large change in connectivity in general, is to maintain and propagate a list of a system's connected nodes [2, 25, 67]. If two lists for the same system are sufficiently different, the system is likely partitioned. Unfortunately, this direct approach is, in most cases, not tractable: it is likely to incur high overheads in large systems, both in terms of memory usage, bandwidth consumption, and hence energy consumption, a prime limiting factor in MANETs.

To overcome this difficulty, our approach (which we term *MtG* for Mind-the-Gap) replaces the explicit representation of reachable nodes by an implicit and potentially inaccurate but compact *summary* of a system's connectivity. More precisely, each node of a system repeatedly constructs a summary of the currently reachable network, and nodes conclude to a partition when two summaries about the same system differ markedly.

Constructing summaries

To be practical and scalable, summaries should ideally be accurate, compact, and robust to network delays and interference. Our approach uses fixed-size bit arrays, termed *filters*, which we construct using a wireless gossip aggregation procedure. Since MANETs are dynamic, nodes may join and leave the network over time, and we do not want our filters to contain outdated information, so we periodically reset every filter to a blank slate. The period between two resets alloted to build the summaries is called an *epochs*,

Figure 4.2 illustrates this construction in a small network of nine nodes. Upon initialization, each node is assigned an initial bit array (the node's *signature*) of the size of the summaries to be constructed. This initial filter contains only unset bits except for one of them. The set bit is selected uniformly at random when a node is configured[2].

When an epoch starts, nodes initialize their local filter with their signature (Label ❶, here shown for the nodes A, B, and C). They then broadcast their current filter (Label ❷), and aggregate the filters received from other nodes with OR operations on each bit. This procedure is repeated over multiple asynchronous rounds during an entire epoch (Label ❸). Eventually, provided the system is connected and the epoch is long enough, each node converges to a summary of the currently reachable network [41, 45]. This summary contains the bit-signatures of all participants the local node was able to hear from (Label ❹, for clarity the figure only shows the signatures of nodes A, B, and C).

Detecting partitions

The system summaries constructed by individual nodes can be exploited in two slightly different ways: *(i) Self-Detection* to detect partitions from within a partitioned system, and *(ii) Assisted-Detection* to detect the partition of a monitored system from an external monitoring system.

Self-detection is illustrated in Figure 4.3. Suppose that a partition occurs just after the construction of the summaries of Figure 4.2 when Epoch e ends (Label ❺). The summaries constructed by each node during Epoch $e+1$ will therefore only encompass signatures from its connected subnetwork (Label ❻). The summary obtained by Node C for Epoch $e+1$ will not contain the signatures of A or B. This summary will thus differ sufficiently from that of Epoch e, allowing C to detect a partition (Label ❼).

Assisted-detection works along the same lines but involves an external monitoring system, and uses discrepancies in space rather than in time. Both types of detection

[2]As a side note, we could have used a hash function on the node identifier to derive a node's signature, in effect constructing a Bloom filter with a single hash function [12]. Bloom filters would have allowed determining that a particular node might have been included in a filter, but as we do not make use of this mechanism here, an initial random bit is both simpler and sufficient.

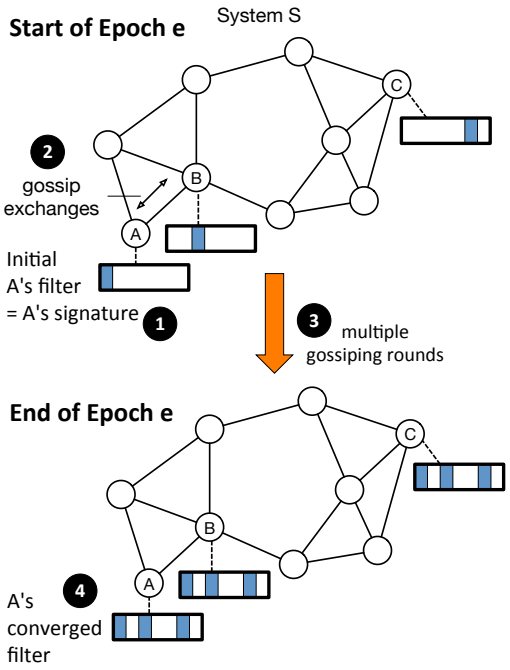


Figure 4.2: Starting from their individual signature, nodes progressively aggregate other nodes' signatures in their local filter. At the end of an epoch, they converge to a summary representing the composition of the subnetwork they have been able to hear from (only the signatures of nodes A, B, and C are shown for simplicity).

are presented more formally in Sections 4.2.2 and 4.2.3.

Parameter trade-offs

So far, we have assumed that the construction of summaries was done perfectly. In practice, two summaries of the same system might diverge for other reasons than a partition. First, individual nodes might crash, leading to small changes in individual summaries. Second, filters might propagate imperfectly over an epoch due to network failures or if an epoch is too short with respect to the network diameter. Such imperfect propagation will lead to variations in the summaries constructed by a same node over successive epochs (used for self-detection), and by different nodes over the same epoch (used for assisted-detection).

Another cause of inaccuracy stems from the compact nature of node signatures and summaries. Signatures can collide, and a partition might only cause small changes in a network's summaries. Consider, in the worst case, a large network using small filters. It is highly probable that all the bits of the summaries will be set to one before and after a large partition. Using a very large filter solves this problem but is a waste of precious resources. The size of system summaries, the length of an epoch, the frequency of gossiping rounds, and the threshold used to detect partitions must therefore be selected with care, depending on the size, dynamism, memory and energy constraints of the system. In the Evaluation section (Section 4.4), we provide pragmatic criteria

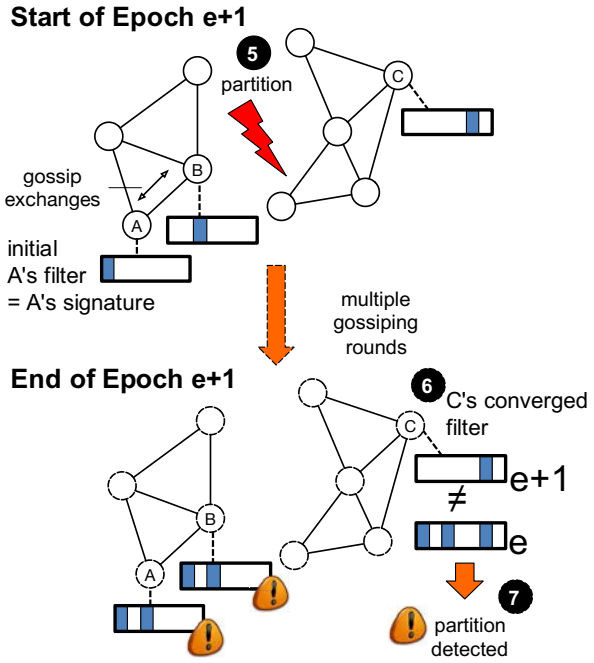


Figure 4.3: When a partition occurs, the summaries between two successive epochs change suddenly, as the signatures of unreachable nodes are no longer aggregated in the converged summary. This sudden change can be detected, and a partition detection event raised.

to set up those various parameters with satisfying trade-offs in common situations, but of course, unusual circumstances may require finer tuning.

Piggybacking and saving resources

Our approach is inherently efficient in terms of memory use, due to the compact and fixed-size filters we use. However, memory isn't the only limited resource in MANETs, and we also have to consider CPU, communications and energy consumption. As detailed in the next subsections, our algorithms consist mostly of simple bitwise operations between filters and integer comparisons, so CPU is not a strained resource. Communications, on the other hand, need a bit more attention. Periodic gossiping can quickly generate a lot of messages and cause a large increase in energy consumption.

Due to the small and fixed size of our filters, we are confident that they can fit in empty bits of network frames generated by other protocols. Such piggybacking allows to save most of the communication costs generated by MtG, but is heavily dependent on the specific stack used, and hard to generalize. For instance, if the routing protocol used by the nodes periodically emits Hello beacon, we can piggyback on those, whereas routing protocols that have a distinct setup phase and do not periodically broadcast during standard operations are much harder to leverage.

As we want our approach to remain independent of any specific protocol stack, we

Constants and functions

Δ_{epoch}	Duration of an epoch.
γ	Threshold used to detect a partition.
$\text{hdist}(s_1, s_2)$	Hamming distance between the bit arrays s_1 and s_2.
$\text{PARTITION}(i, e)$	Event representing a partition in system i at epoch e.

Variables maintained by a node p_i in a monitored system

$sysID_i$	The ID of the system the node p_i belongs to.
clock_i	p_i's local clock.
$epoch_i$	p_i's current epoch number.
$node\_sig_i$	The one-bit signature of p_i.
$filter_i$	The system summary being constructed by p_i
$sum_i[]$	An array of the system summaries observed by p_i at the end of each past epoch, indexed by epoch numbers (used for self-detection)

Variables maintained by a node p_i in a monitoring system

$sumSet_i$	The set of summaries propagated to the monitoring node. $(id, ep, s) \in sumSet_i$ means that a node from system id generated a system summary s at the end of epoch ep, and that p_i is aware of this summary.

Table 4.1: MIND-THE-GAP: Notations and variables maintained by each node

did not consider this aspect in details in the following, but we acknowledge it is an important step before a real world deployment is possible. We surmise that in most situations, an efficient implementation in terms of communication and energy costs is possible.

In the following, we first detail the self-detection variant of our algorithm (Section 4.2.2), before moving on to the case of assisted-detection (Section 4.2.3).

4.2.2 Self-detection protocol (MtG/Self-detect)

This first variant, termed *MtG/Self-detect*, allows a system to monitor its own evolution over time to detect when it becomes partitioned. The protocol is described by Algorithms 5 and 6. Table 4.1 provides a summary of the variables and notations used.

When a node p_i starts participating to the system, it does not take part in the current epoch (its $epoch_i$ variable is set to $\perp$), and waits for the next epoch to start at time $\left(\left\lfloor \frac{\text{clock}_i}{\Delta_{\text{epoch}}} \right\rfloor + 1\right) \times \Delta_{\text{epoch}}$ before joining the protocol, where Δ_{epoch} is the duration of an epoch, and clock_i represents p_i's local clock. We assume that clocks are loosely

Algorithm 5: MtG/Self-detect: Filter aggregation (at p_i)

```
1 every X seconds do
2     BROADCAST AGG⟨sysID_i, epoch_i, filter_i⟩
3 on receive AGG⟨sysID, epoch, filter⟩ do
4     if epoch_i = epoch and sysID_i = sysID then
5         filter_i ← OR(filter_i, filter)
```

Algorithm 6: MtG/Self-detect: Change of epoch (at p_i)

```
1 on expiration EPOCH_TIMER do
2     sum_i[epoch_i] ← filter_i
3     for t ∈ 0..epoch_i − 1 do
4         if hdist(sum_i[epoch_i], sum_i[t]) > γ then
5             raise PARTITION(sysID_i, epoch_i)
6     filter_i ← node_sig_i                    ▷Resetting p_i's filter
7     epoch_i ← ⌊clock_i / Δ_epoch⌋             ▷New epoch
8     set timer EPOCH_TIMER at (epoch_i + 1) × Δ_epoch
```

synchronized between all nodes, with a drift remaining small compared to the duration of an epoch Δ_{epoch}, so that all nodes in the MANET work for the same epoch during a large fraction of Δ_{epoch}. The code of the joining mechanism is not shown in the presented pseudo-code for the sake of clarity.

We assume that nodes know the identifier of the system they belong to, and that they can ignore messages sent by nodes belonging to other systems, if necessary. When a node actively participates in an epoch, it periodically broadcasts its current filter (lines 1-2 of Alg. 5). When it receives a neighbor's filter for the system it belongs to, and for the epoch it currently participates in (lines 3-4), it incorporates the received filter into its own filter by using a logical OR over the two bit fields (line 5). Otherwise, the message is simply dropped. This simple process implements a push-based aggregation [41], and is robust and efficient.

At the end of an epoch, each node stores its final filter as the system summary for this epoch (line 2 of Alg. 6) and compares it to prior summaries by calculating the Hamming distance between the two filters (hdist$(-,-)$, line 4), i.e. counting the number of bits in which they differ. The current filter is then reset (line 6) and a new aggregation epoch starts (lines 7-8).

If there is "enough" difference between filters (using the threshold γ at line 4), this is a sign of significant change in the MANET over the corresponding time interval and a partition is detected (with the PARTITION event at line 5). The main difficulty, and

Algorithm 7: MtG/Assisted: Change of epoch at a node p_i belonging to a monitored system

```
1 on expiration EPOCH_TIMER do
2     BROADCAST SUMMARY⟨sysID_i, epoch_i, filter_i⟩
3     filter_i ← node_sig_i                          ▷resetting p_i's filter
4     epoch_i ← ⌊clock_i / Δ_epoch⌋                  ▷New epoch
5     set timer EPOCH_TIMER at (epoch_i + 1) × Δ_epoch
```

an important contribution of this work, is to determine the proper threshold γ for a *big enough difference* between filters. This relies on hypotheses on the MANET evolution and is examined in more details in Section 4.3.

4.2.3 Assisted-detection protocol (MtG/Assisted)

The second variant (*MtG/Assisted*) allows an assisting system to monitor our target system in order to detect partitions in it. The protocol is detailed in Algorithms 7 and 8. The main idea consists in propagating *within the monitoring system* summaries constructed by nodes of the *monitored* system. This propagation makes it then possible to detect whether two nodes from the monitoring system have observed a large enough discrepancy in two summaries of the same monitored system for the same epoch, hinting at a partition.

More specifically, nodes in the target system execute the same aggregation gossip as previously (cf. Alg. 5). At the end of an epoch, however, the nodes do not store the signature but instead broadcasts them with a SUMMARY message (line 2 in Alg. 7).

When a node from the monitoring system receives a SUMMARY message (line 1 in Alg. 8), it stores it (line 3). The monitoring system then executes its own gossip aggregation *of the target system's signatures* with SUMMARY_SET messages (lines 5-10 in Alg. 8). When a node receives a SUMMARY_SET message, it stores the new summaries (line 9) and then checks if it has two different enough summaries from the same epoch and for the same target system, indicating a potential partition (procedure CHECKPARTITION(), lines 11-15).

4.3 Analysis

4.3.1 System Model

We consider a mobile area network (MANET) consisting of a set S of n nodes communicating through wireless channels. At each epoch, each node n_i initializes and updates a filter of size f. At the start of the epoch, each node sets the bits of its filter to 0 except for its own signature bit $node\_sig_i$ which is set to 1. This signature bit does

Algorithm 8: MtG/Assisted: Signature aggregation at a node p_i belonging to a monitoring system

1 **on receive** $SUMMARY\langle sysID, epoch, filter\rangle$ **do**
2 **if** $sysID_i \neq sysID$ **then**
3 $sumSet_i \leftarrow sumSet_i \cup \{(sysID, epoch, filter)\}$
4 CHECKPARTITION()
5 **every** Y *seconds* **do**
6 BROADCAST SUMMARY_SET$\langle sysID_i, sumSet_i\rangle$
7 **on receive** $SUMMARY\_SET\langle sysID, sumSet\rangle$ **do**
8 **if** $sysID_i = sysID$ **then**
9 $sumSet_i \leftarrow sumSet_i \cup sumSet$
10 CHECKPARTITION()
11 **procedure** CHECKPARTITION() **is**
12 $P \leftarrow \left\{ (i, e, s_1, s_2) \mid \begin{array}{r} \text{hdist}(s_1, s_2) > \gamma \ \wedge \\ \{(i, e, s_1), (i, e, s_2)\} \subseteq sumSet \end{array} \right\}$
13 **For all** $(i, e, s_1, s_2) \in P$ **do**
14 $sumSet_i \leftarrow sumSet_i \setminus \{(i, e, s_1), (i, e, s_2)\}$
15 **raise** PARTITION(i, e)

not change between epochs. Hence, even with unique identifiers, the identity of each node is reduced to an integer between 1 and f chosen uniformly at random. During the aggregation phase, each node gossips its filter to its neighbors in asynchronous rounds and updates it with logical OR operations as it receives filters from its neighbors. The resulting filter at the end of the epoch is used as a (potentially imperfect) *summary* of the system reachability for that node in that epoch. Using summaries for consecutive epochs $sum_i[e]$ and $sum_i[e+1]$, each node n_i in the self-detection scenario wishes to answer the following question: "Do the summaries $sum_i[e]$ and $sum_i[e+1]$ indicate the presence of a new network partition, and with what level of confidence?" For this purpose, we define in Algorithms 6 and 8 a *partition threshold* γ beyond which protocols trigger mitigating measures. This threshold is a bound on the Hamming distance $\text{hdist}(sum_i[e], sum_i[e+1])$ between both summaries. Please note that the question in the assisted-detection scenario is similar, but must be answered by the monitoring system. Furthermore, the monitoring system also has the luxury to use summaries built by different nodes for the same epoch, which is not possible in a self-detecting network. To simplify the discussion, for the rest of this section we assume that we are in the self-detection scenario.

4.3.2 Operating in paradise (ideal conditions)

If the network is perfect (i.e., no churn, no noise) and the aggregation is sufficiently long, all the summaries of the network (or of each subnetwork if there is a partition) are identical at the end of each epoch. In this (highly unrealistic) scenario, if the Hamming distance between summaries $sum_i[e]$ and $sum_i[e+1]$ is anything but zero, we can be sure that said filters come from different subnetworks caused by a partition, and we can use the threshold $\gamma = 0$ in Algorithms 6 and 8.

However, even if the Hamming distance between two summaries is exactly zero, they may still come from different systems due to collisions in the random choice of the original node signatures. We now evaluate the likelihood of such a scenario. Let $B(sum_i[e])$ be the number of nonzero bits in a filter (summary) of size f when inserting n randomly-selected bits (node signatures) into it. The expected number of bits set to '1' in the filter is

$$f \cdot \left(1 - \left(1 - \frac{1}{f}\right)^n\right)$$

.

More precisely, it was shown in [6] that $\mathbb{P}(B(f_e) = j)$ is given by the formula

$$\mathbb{P}(B(sum_i[e]) = j) = \frac{\binom{f}{j} \cdot j! \cdot \left\{ {n \atop j} \right\}}{f^n}$$

where $\left\{ {a \atop b} \right\}$ stands for the Stirling numbers of the second kind [36]. Stirling numbers are gruesome to handle and are usually tackled asymptotically, but since in this work we are ultimately interested in efficient implementations using small filters, we can easily resort to mathematical simulations. As n and f increase, the distribution of $B(sum_i[e])$ is sharply concentrated around its expected value [64]. At first sight, this is disappointing: for instance with $f = 32$ and $n = 64$ the expected number of '1's in the summaries is 27.8. However, if two subnetworks of size n are disjoint following a partition, one can easily derive that the expected size of the Hamming distance of their respective summary is

$$2 \cdot f \cdot \left(1 - \frac{1}{f}\right)^n \cdot \left(1 - \left(1 - \frac{1}{f}\right)^n\right)$$

.

Figure 4.4 shows the distribution of the Hamming distance of the summaries for two disjoint subnetworks of 64 nodes with filters of size 32. In this example, the probability that the Hamming distance is zero is approximately 10^{-5}. Note that this corresponds to a network of 128 nodes partitioned into exactly two partitions of 64 nodes.

Two partitions of equal size is the worst possible scenario since the probability that both summaries are identical decreases quickly as the size of each subnetwork diverges. Note that in the self-detection variant, there is no central authority that can compare

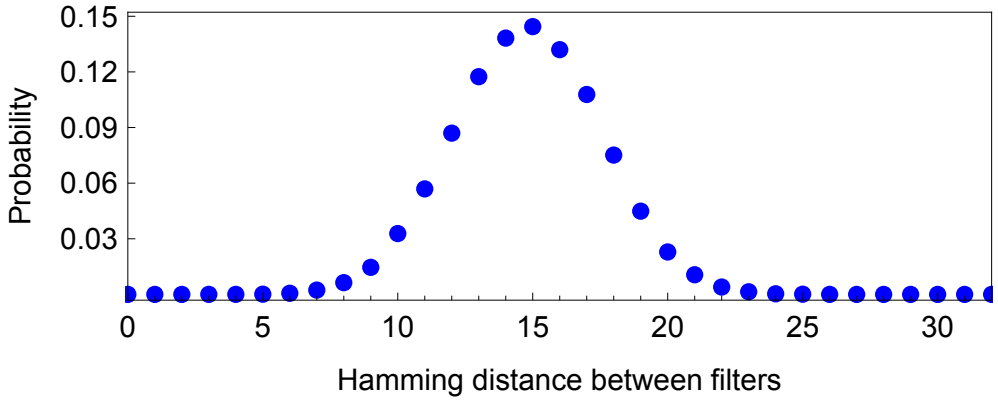


Figure 4.4: Distribution of the Hamming distance of two summaries of size $f = 32$ with $n = 64$ inserted signatures coming from independent subnetworks.

the summaries from both partitions; if the size of the partitions greatly differs, the small subnetwork will easily detect the partition and trigger mitigating measures. This is further discussed in the next section.

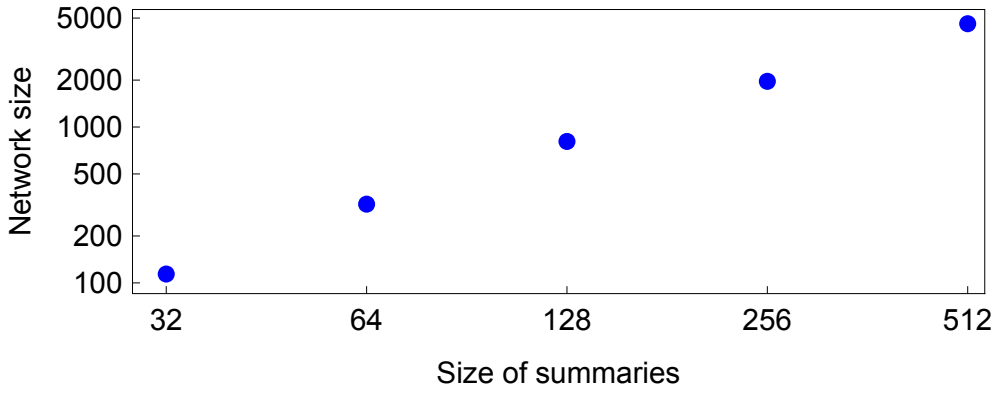


Figure 4.5: Maximum network size for a probability of false negative of 10^{-5} when partitioned into two partitions of equal size for different summary sizes.

Figure 4.5 shows, in this idyllic scenario without noise and churn, the maximum network size allowing a probability of false negative of 10^{-5} when partitioned into two partitions of equal size for different summary sizes (again this is the worst possible scenario). Under perfect network and convergence assumptions, MtG can thus easily handle partition detection with high accuracy in networks of 128 nodes or less with 32-bit summaries, in networks of 800 nodes or less with 128-bit summaries, and in networks of 4500 nodes or less with 512-bit summaries.

Note that the size of the summary grows sub-linearly with the number of nodes n in the network. This may look surprising at first glance, since it means we have more and more collisions between individual node signatures as the size of the network grows, but can in fact be easily explained. Indeed, we do not want to detect the presence of individual nodes and we do not care about individual collisions, we just need to

distinguish summaries from different sub-networks. As errors essentially happen when the summaries are both entirely filled with '1's, a filter of size f can be used for a network of size $n < f.log(f)$, since this is equivalent to the coupon collector problem. Or, the other way around, a network of size n needs a filter of size $f > n/log(f)$, which is sub-linear in n as expected.

4.3.3 Operating in hell (imperfect aggregation and dynamic networks)

In a real deployment, of course, filters may diverge within the same subsystem due to churn and node crashes. They may also diverge due to the imperfect aggregation caused by network contention, node mobility or a hostile environment. One should thus set the threshold γ to a strictly positive value to account for these differences between epochs, otherwise the nodes or the monitoring system will constantly and uselessly trigger mitigating measures under normal network operation. Modeling these constraints in MANETs separately is challenging to say the least, however it is fortunately unnecessary in this work as they indistinguishably all result in inaccuracies in the node-constructed summaries. Instead, we define a generic churn parameter c and assume that there are c nodes in epoch e that fail in epoch $e+1$, and conversely that there are c nodes that appear in epoch $e+1$ but failed in epoch e. A failing node simply fails to disseminate its signature to other nodes in the network.

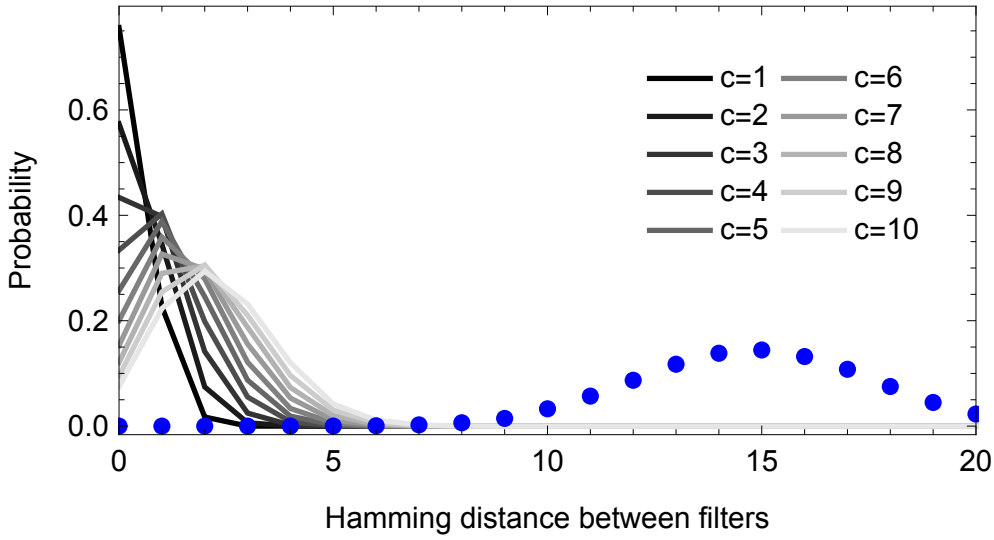


Figure 4.6: Distribution of the Hamming distance of two summaries of size $f = 32$ with $n = 64$ inserted signatures and churn parameter $c \in \{1, 2, 3, \ldots, 10\}$ (gray curves). The blue points correspond to two partitioned systems of 32 nodes. The probability mass functions are calculated with 10^5 randomized experiments.

Figure 4.6 shows the distribution of the Hamming distance of two summaries of size $f = 32$ used by a system of $n = 64$ nodes. Each gray curve corresponds to a churn

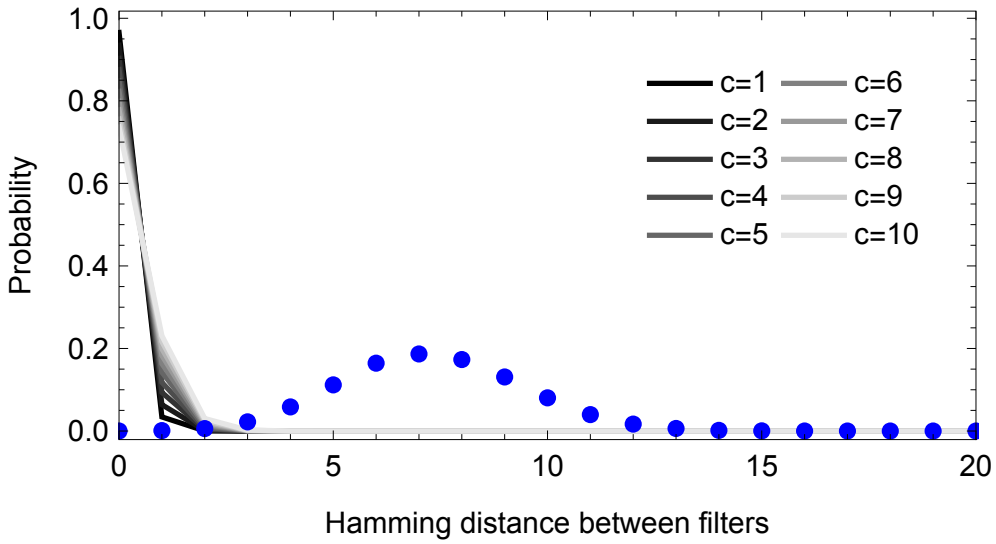


Figure 4.7: Distribution of the Hamming distance of two summaries of size $f = 32$ with $n = 128$ inserted signatures and churn parameter $c \in \{1, 2, 3, \ldots, 10\}$ (gray curves). The blue points correspond to two partitioned systems of 64 nodes. The probability mass functions are calculated with 10^5 randomized experiments.

level c varying from 1 to 10. The blue points correspond to two partitioned systems of 32 nodes. The gray and blue probability mass functions are almost disjoint, thus if we set the partition threshold at $\gamma = 7$, we essentially catch 100% of the partitions and no normal operational event. In figure 4.7, we repeat the process with the same filter size but with systems of $n = 128$ nodes. There is a small overlap between the gray and blue curves, but by setting the partition threshold to $\gamma = 2$, we still detect 99% of the worst possible partitions and treat less than 1% of the normal events at $c = 10$ as partitions.

We emphasize that these results are conservative for three reasons. First, $c = 10$ and $n = 64$ corresponds to a churn rate of 15% per epoch. The operator of a network in such an environment might want to trigger mitigating measures even in the absence of a partition. Second, it assumes that the network is partitioned in two pieces of equal size. As mentioned earlier, this is the most pessimistic scenario (i.e., it will systematically yield the lowest Hamming distances between summaries for the different subsystems). Third, a new partition will generate filters of unequal size between epochs, a fact that we do not even leverage in this work. It is thus clear that like for many applications of bit fields as a compact representation, such as Bloom filters [19], the efficiency of our approach is uniquely determined by the size of the filters and the number of nodes in the network: with very small filters and very large networks, the filters, even once partitioned equally, will be filled with '1's. Like for our idyllic scenario, we can very easily tackle networks of size 128 with 32-bit filters, and networks of size 2,048 with 256-bit filters.

4.4 Evaluation

We evaluate our proposal along three dimensions. We first investigate whether filters can effectively distinguish between joined and partitioned networks (Section 4.4.2). We then assess the ability of MtG protocols to detect partitions and compare them to two state-of-the art approaches (Section 4.4.3). Finally, we explore the influence of various parameters in Section 4.4.4 and we suggest concrete values for satisfying trade-offs in Section 4.4.5.

4.4.1 Experimental setup and metrics

Unless stated otherwise, we configure MtG to use 32-bit filters, asynchronous rounds of 0.3 second, and epochs of 5 seconds (i.e., up to 16 rounds per epoch). Nodes are set up with a 100m communication range, and positioned over a 400m×400m area. The exact number of nodes and their positions depend on the experiment, as we detail below.

We use the Omnet++/Inet framework [76] for our simulations. Each experiment is repeated 10 times, with different random seeds for nodes signatures and positions. Presented results are averages over the 10 runs. We do not show error bars as they are negligible. We use the following metrics:

- **The normalized maximum pair-wise internal distance** (*internal distance* for short) measures the maximum Hamming distance between the filters being maintained by two nodes of the same monitored system S, normalized by the size of the filters f (32 in our experiments). When S is connected, this distance should converge to zero at the end of each epoch. Formally, we have

$$\text{internal\_d}(S) = \max_{(p_i,p_j)\in S^2} \left(\frac{\text{hdist}(\mathit{filter}_i, \mathit{filter}_j)}{f} \right).$$

- **The normalized minimum pair-wise inter-partition distance** (*external distance* for short) measures the minimum Hamming distance between the filters of two partitions S_1 and S_2 of a system, normalized by the size of the filters. When S_1 and S_2 are disconnected, this distance should remain above the γ threshold, even in a converged state, in order to distinguish a partition from a connected configuration. Formally we have

$$\text{external\_d}(S_1, S_2) = \min_{\substack{(p_i,p_j)\in \\ S_1\times S_2}} \left(\frac{\text{hdist}(\mathit{filter}_i, \mathit{filter}_j)}{f} \right).$$

- **The error rate** is the number of nodes belonging to a partitioned system that do not detect the partition (*false negatives*) summed with the number of nodes belonging to a fully connected system that raise a partition alert (*false positives*), over the total number of nodes. In the vast majority of cases, the error rate is

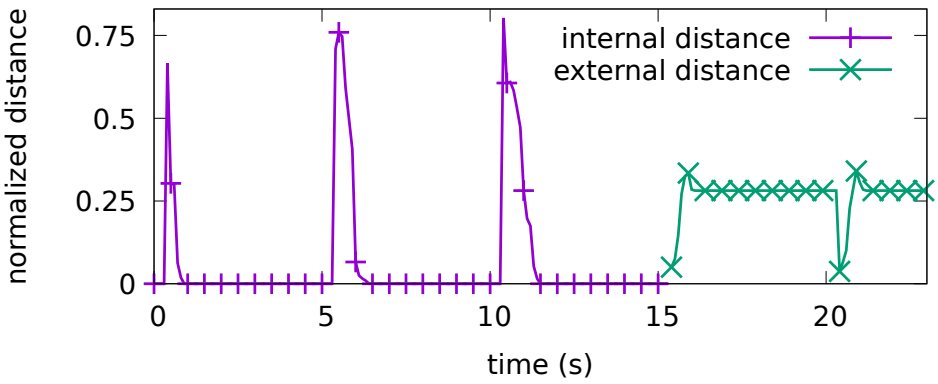


Figure 4.8: Filters are an effective representation of a network membership. They converge when the system is connected, and they are clearly distinct when there is a partition.

equivalent to the number of false negatives. Indeed, we may get false negatives due to collisions in filters and disconnected subsystems may obtain identical filters; but the gossip convergence of filters is extremely robust and nodes in a non-partitioned system always converge towards the same filter under realistic circumstances.

- **The per-node per-round bandwidth** (*bandwidth* for short) represents the amount of information sent by each node per round, measured in bits.

4.4.2 Effective representation

We set up a system S of 120 nodes divided into two groups S_1 and S_2 of 60 nodes each. S_1 and S_2 are initially distributed over the same 400m×400m area, but drift apart in opposite direction at 25 m/s, S_1 heading North and S_2 South. S_1 and S_2 thus become unable to reach each other at around 15 seconds into the experiment (end of the 3rd epoch). We monitor over time the internal distance of S and the external distance between S_1 and S_2. Figure 4.8 shows that before the partition (0-15 seconds), the internal distance goes down to 0 very rapidly, in less than 2 seconds (6 rounds). Once the partition occurs (15-24 seconds), the external distance between the two partitions is always strictly positive, and converges quickly to a value noticeably above 0.

These two results combined demonstrate that filters are efficient to represent a system membership, in a quick, accurate, and compact manner. Even after only a few rounds, all nodes in a system agree on the same filter, while disconnected subsystems have very distinct filters, all the while using 32-bit filters for systems containing over a hundred nodes.

4.4.3 Partition detection

In this second set of experiments we compare the filters used by MtG with those of two baseline approaches:

- **The *graph-coloring* baseline:** Each node randomly choses a 16-bit integer, a *color*, and gossips it. When it receives a color from another node, it keeps the larger one. If a network is not partitioned, all nodes will eventually agree on the same color; if there is a partition, each subsystem will agree on a different color. Hence, the color can serve as a simple way to distinguish subsystems.
- **The *full-list* baseline:** Each node maintains an exhaustive list of all node identifiers it has encountered, and gossips it around. At the end of an epoch, all connected nodes agree on the same list, which is the membership list of their subsystem. If two different lists are observed, it is a sign of a partition.

We repeat the setup of the previous subsection: two 60-node groups moving away from each other. We run four experiments in which nodes execute either one of the variants of MtG or one of the two other protocols.

- In the first experiment, all 120 nodes execute MtG/Self-detect, the self-detection version of our approach.
- In the second one, we add a third group of nodes from an independent system in the middle, serving as the monitoring group, and we run MtG/Assisted, the third-party version of our protocol. The third group is constituted of 20 fixed nodes, set up to ensure the coverage of the whole area of the experiment.
- In a third experiment, we used the graph-coloring baseline instead of our proposed filters to represent the network membership, in self-detect mode.
- Finally in the fourth experiment, we used the full-list baseline instead of filters, in self-detect mode.

The results are summarized in Table 4.2. The bandwidth consumption assumes the following: filters are 32 bits long, node identifiers use 32 bits (size of an IPV4 address), and colors are 16-bit integers.

In both modes, our protocols accurately detect the partition when it occurs, with very modest bandwidth consumption, comparable to the graph-coloring approach. In contrast, the graph-coloring approach only detects the partition for half of the nodes. Indeed, the nodes in the group of the "winning" color do not see any change when the partition occurs.

The extensive-list approach detects partitions accurately, but its bandwidth consumption is orders of magnitude larger than with our approach, even with a system with as little as 120 nodes. Moreover, this bandwidth usage varies considerably over the execution of the protocol, with a low bandwidth usage at the beginning of an epoch (using small lists) and an uncontrolled usage as lists are aggregated to the full

	Error rate	Bandwidth (bits sent/round.node)	
		average	max
graph-coloring	**50%**	16	16
full list	0%	**1995**	**3840**
MtG (self-detect)	0%	32	32
MtG (monitored node)		32	32
(monitoring node)	0%	32	64

Table 4.2: Partition detection performance.

membership, e.g. up to 1,920 bits for a 60-node group, or 3,840 for a 120-node group. This bandwidth usage is orders of magnitude higher than with the filters, and increases asymptotically faster as the network grows. Furthermore, it is in the expected common case when the lists are converged and that there is no partition that the usage is the highest, which is the opposite of the desired behavior.

4.4.4 Pushing the limits

We now explore specific adversarial conditions that MtG may encounter, and suggest how to work around them.

First, the length of the epochs is a tradeoff between the quality of convergence between filters and the freshness of the information they contain. While a long epoch provides better convergence guarantees, shorter epochs avoid keeping information about crashed or unreachable nodes, and faster mitigation following partitions. The length of the epochs needs to be adapted to the size of the considered systems. We set up various experiments with system of increasing sizes. We keep the density of nodes constant between all experiments, simply distributing a larger number of nodes over a larger area, resulting in systems of increasing diameters, from 4 to 16 hops. The length of rounds and epochs is also constant between experiments. Figure 4.9 shows the evolution of the internal distance of each system over time. One can clearly see that as the diameter increases, the convergence is slower, and past a certain diameter, filters do not have enough time to converge anymore, as the distance does not reach zero.

Given the number and type of devices deployed in a MANET and the physical area over which they are deployed, a user can get a rough estimation of the system's diameter: $D \approx \sqrt{N} \approx C/r$ where N is the number of devices, C is the physical diameter of the area covered, and r is the communication range of the devices. If the two estimations are noticeably different, it is usually a sign that the number of devices deployed is not adapted to the area covered. We recommend to set the length of an epoch (in rounds) to twice the expected maximal diameter of the system (in hops).

Finally, we test the resilience of MtG to bad communication conditions and message loss. We set up a system of 120 nodes distributed in two groups of 60 nodes each and make them drift apart under the same scenario described above. In order to stress the system, we reduce the length of the epoch to just 6 rounds, and we tune the drifting speed so that a partition occurs at the end of the second epoch. MtG should hence raise a partition alert during epoch 3. Figure 4.10 shows the fraction of nodes that raise an alert during each epoch. With 20% of messages loss, everything works perfectly well. With 40% loss, barely 10% of nodes confuse the bad communication conditions with a real partition in epoch 2. We need to reach an enormous level of 60% loss before a noticeable fraction of MtG nodes raise a false alert before the partition actually occurs. We argue that under such harsh conditions, a system operator has more pressing issues than partitions, and will want to trigger available counter-measures just as if a real partition was occurring. This raises our confidence that MtG is resilient to message loss under all realistic circumstances.

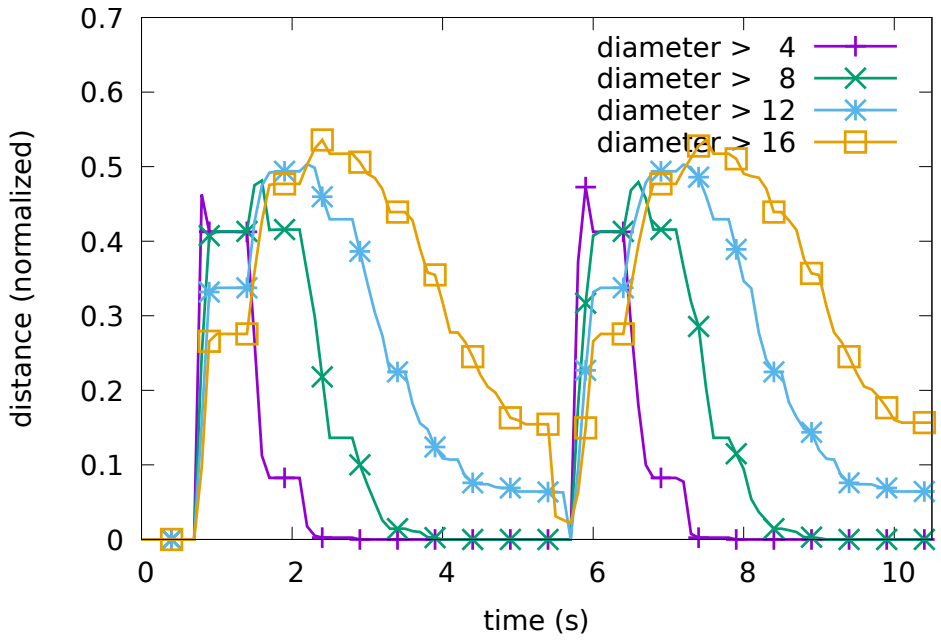


Figure 4.9: With fixed epoch and round durations, the larger the network is, the longer it takes to converge. If the network is too large, the filters do not converge anymore, so it is important to adapt the length of an epoch to the size of the network.

4.4.5 Concrete parameters settings

Based on the results of our simulations, we suggest for a network operator wishing to use MtG to set-up the following parameters:

- round: if using piggybacking, this is a constraint and not a free parameter; if **not** piggybacking, a round should be as short as possible without excessively

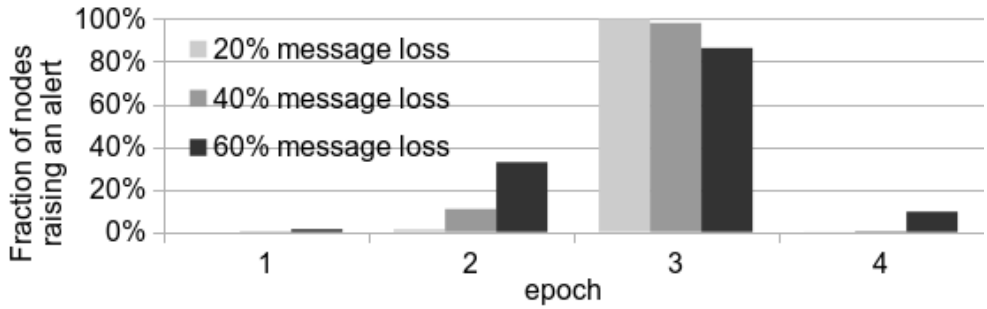


Figure 4.10: Fraction of nodes answering detecting a partition, under various levels of message loss. The real partition occurs at the end of epoch 2. Our protocol is resilient to message loss, but when the the level of disruption is really high, it starts to confuse message loss with partition.

flooding the waves or taxing the energy resources

- epoch: In number of rounds, an epoch should be **at least** one and a half as long as the diameter of the network. A good approximation to use when the actual network diameter is unknown is: diameter of the area covered by the network over communication range of the devices involved. In absolute duration, an epoch must be short enough that a node participating at the beginning of an epoch is still in communication range at the end of the same epoch, so we suggest trying to keep an epoch **at most** twice the time it takes for a device to cover its own communication range. The first constraint is by far the most important of the two.
- filter width: for a network of n nodes, a good approximation without resorting to complex mathematical analysis is to use around $n/8$ bits for the filter; if using empty bits from another protocol frames, use as many bits as available, as long as it is greater than $n/8$.
- detection threshold: $\gamma = 2$ is a reasonable value for most realistic circumstances with current networks.

4.5 Conclusion & Future Work

In this chapter, we presented Mind-the-Gap, a lightweight method to detect partitions in a MANET, either by the nodes forming the MANET themselves or by an external supporting system. Our analysis and evaluation show the ability of our approach to detect such partitions even under aggregation imperfection and imperfect networking conditions.

For future work, we are interested in developing a more formal method to derive the partition threshold γ based on the filter and network sizes, or at least to provide these thresholds for a larger number of parameters. In particular, as mentioned in Section 4.3 we do not leverage the number of bits set to '1' in our summaries. When

a system is partitioned, two general cases can occur. If both partitions have the same size, the summaries from one epoch to the next will very likely contain significantly less bits set to '1'. If one partition is small and the other is large, the large partition might view its evolution as a normal churn event, but the small partition will see an even higher reduction in its number of summary bits set to '1' between consecutive epochs. Considering the number of bits set to '1' may allow us to tackle higher noise levels, as well as larger systems with good accuracy, without increasing the filter size. It would also avoid triggering an alert when a large number of nodes join the network or when two partitions reconnect, since this events *increase* the number of '1' in the filters, but are still detected with our current approach as large changes in membership.

Mind-the-Gap demonstrates that it is possible to use epidemic protocols to make a distributed system aware of its circumstances and able to detect changes in its environment, even with only local information, in an efficient and resilient manner. Building upon this work, it is likely possible to then react and adapt to the detected changes in a self-adaptive fashion and move toward the opportunistic systems composed of smarter basic blocks we argued for in Chapter 1. In the next chapter, we will focus on the fundamentals of gossip protocols and show how to extend their application to new problems and new contexts.

Chapter 5

Extending traditional gossip: HyFN Decentralized construction of KFN graphs

In Chapter 1, we presented a two-fold vision: a holistic approach to distributed systems relying on opportunistic systems composed of smart basic blocks. In Chapters 3 and 4, we explored the two faces of this vision, and in both cases we used gossip protocols to implement our proposals. In this chapter, we go deeper into gossip and show that their field of application is even wider than it looks at first glance.

Indeed, all traditional gossip-based self-organizing overlays assume some level of transitivity in the neighborhood relationship, but we propose a simple extension that still functions even when such fundamental assumptions do not hold anymore, thus demonstrating that gossip protocols are very adaptable and great candidates to solve the many challenges faced by complex distributed systems nowadays.

5.1 Motivation

Epidemic or gossip protocols, thus named because they replicate how rumors and diseases propagate, are a very efficient and well known approach to disseminate information in a distributed system. Notably, they are the current best method to build k-Nearest-Neighbor (KNN) graphs. This type of graphs, which groups similar nodes together, has found usage in a number of domains, including machine learning, recommending system, and search optimization.

Some applications do not however require the k closest nodes, but the k most dissimilar nodes, what we term the k-Furthest-Neighbor (KFN) graph. This is especially the case for applications that try to find *complementary* profiles which need to collaborate on a common task, and it is consequently an important problem to solve for opportunistic distributed systems, but also for other collaborative environments such as mutualized data-centers.

For instance, Virtual Machines (VMs) placement —i.e. the (re-)assignment of workloads in virtualised IT environments— would be a good application for KFN. The problem consists in finding an assignment of VMs on physical machines (PMs) that minimises some cost function(s) [70]. The problem has been described as one of the most complex and important for the IT industry [9], with large potential savings [50]. An important challenge is that a solution does not only consist in packing VMs onto PMs — it also requires to limit the amount of interferences between VMs hosted on the same PM [83]. Whatever technique is used (e.g. clustering [52]), interference aware VM placement algorithms need to identify complementary workloads — i.e. workloads that are dissimilar enough that the interferences between them are minimised. This is why the application of KFN graphs would make a lot of sense: quickly identifying complementary workloads (using KFN) to help placement algorithms would decrease the risks of interferences.

However, if the construction of KNN graphs in decentralized systems has been widely studied in the past [42, 78, 32], those works do not transfer easily to KFN graphs, because existing approaches typically assume a form of "likely transitivity" of the similarity between nodes: if A is close to B, and B to C, then A is likely to be close to C. Unfortunately this property no longer holds when constructing KFN graphs. Consequently these approaches, as demonstrated in the remainder of this chapter, are not working anymore when applied to this new problem.

To address this challenge, we propose HyFN (standing for *Hybrid KFN*, pronounced *hyphen*), an hybrid distributed approach for the decentralized construction of k-furthest-neighbor graphs. We show that HyFN is able to construct a KFN graph with 3200 nodes in less than 17 rounds, when a traditional greedy approach is unable to converge. We also show that our proposal is highly scalable, with a convergence time evolving in $O(log(n))$ for larger graphs.

The remainder of this chapter is organized as follows: after a brief reminder on k-nearest-neighbor (KNN) graphs and their decentralized construction in peer-to-peer networks, we then present our intuition for the construction of a k-furthest-neighbor graph (KFN) in Section 5.2. In Section 5.3, we describe in more detail HyFN and its variants. We evaluate our approach in Section 4.4, and conclude in Section 5.5.

5.2 Decentralized Construction of a KFN graph

5.2.1 Background: Decentralized KNN Graph Construction

The problem of constructing a k-furthest-neighbor (KFN) graph can be seen as a variant of a k-nearest-neighbor (KNN) graph construction that uses an opposed similarity. We already presented KNN construction in Chapter 2, but we will go for a brief reminder here.

The principle of a typical P2P protocol for KNN graph construction [28, 78] is

Algorithm 9: Greedy decentralized KNN algorithm executing at node p

```
1 each round do
2     q ← one random neighbor from Γ(p)
3     send ⟨PUSH, Γ(p) ∪ {p}⟩ to q ; request Γ(q) from q        ▷ push - pull
4     cand ← Γ(p) ∪ Γ(q) ∪ {r random nodes} \ {p}
5     Γ(p) ← argtop^k_{g∈cand} (sim(p, g))

6 on receiving ⟨PUSH, Γ′⟩ do
7     cand ← Γ(p) ∪ Γ′ \ {p}
8     Γ(p) ← argtop^k_{g∈cand} (sim(p, g))
```

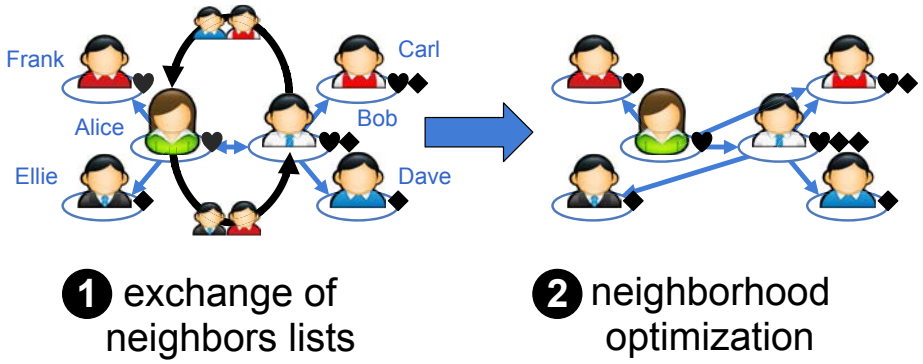


Figure 5.1: A round of greedy decentralized KNN construction

shown in Algorithm 9, in its push-pull variant[1]. Starting from a random neighborhood, individual nodes repeatedly select a random neighbor q (line 2), exchange their current neighborhood with that of q (noted $\Gamma(q)$, line 4), and use the gained information to select more similar neighbors (line 5)[2]. Similarly, when receiving a new neighborhood pushed to them, nodes update their local view with the new nodes they have just heard of (lines 6-8). The intuition behind this greedy procedure is that if A is similar to B, and B to C, C is likely to be similar to A as well. To avoid local minima, this greedy procedure is often complemented with a few random peers (returned by a *peer sampling service* [43], tuned with parameter r at line 4).

This mechanism is illustrated in Figure 5.1. In this example, node *Alice* is interested in hearts (*Alice*'s profile), and is currently connected to *Frank*, and to *Ellie*. During this round, Alice selects *Bob* as her exchange partner. After exchanging her neighbors list with *Bob*, *Alice* finds out about *Carl*, who appears to be a better neigh-

[1]The presented model is close to the *Vicinity* algorithm [78], but variations exist, most notably the T-Man algorithm [42], which buffers and selects nodes differently.

[2]argtop^k returns a k-tuple of nodes that maximizes the similarity function $\mathrm{sim}(p, -)$. Said differently, argtop^k generalizes the concept of argument of the maximum (argmax for short) to the k top values of a function over a finite discrete set.

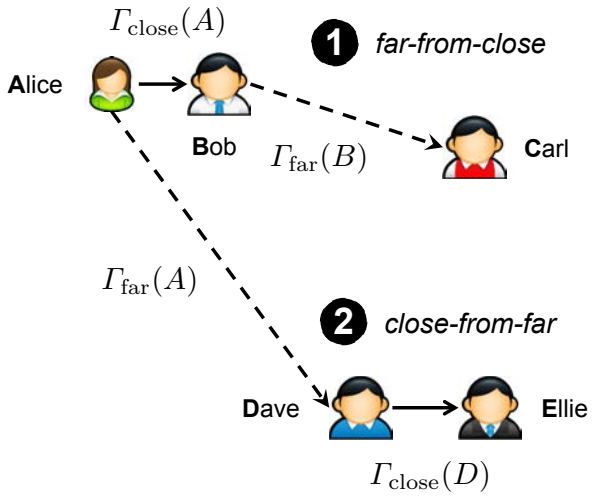


Figure 5.2: The two heuristics we propose to construct a KFN graph

bor than *Ellie*. As such, *Alice* replaces *Ellie* with *Carl* in her neighborhood. Similarly *Bob* detects that *Ellie* is a better neighbor than *Alice*, and drops *Alice* in favor of *Ellie*.

5.2.2 Moving to Decentralized k-furthest-neighbor Graph Construction

Algorithm 9 can be easily adapted to compute a decentralized k-furthest-neighbor (KFN) graph by using a negative similarity at line 5:

$$\Gamma(p) \leftarrow \underset{g \in cand}{\overset{k}{\text{argtop}}} \big(-\text{sim}(p, g) \big) \tag{5.1}$$

Unfortunately, with this modification, one of the key premises of Algorithm 9 disappears: the far neighbors of a far neighbor are not so likely to be interesting candidates to construct a KFN graph. Said differently, if A is far from B, and B far from C, this does not imply that A is far from C (or further from C than any other node taken randomly in the dataset).

Starting from this observation, we propose instead to use a *dual strategy* that constructs an *intermediate KNN graph* in order to construct *a final KFN graph*. In our approach, each node p maintains two views containing k nodes each: $\Gamma_{\text{close}}(p)$ and $\Gamma_{\text{far}}(p)$.

$\Gamma_{\text{close}}(p)$ uses the algorithm shown in Algorithm 9 to converge towards the k most similar other nodes in the system. $\Gamma_{\text{far}}(p)$ employs two greedy optimization heuristics that exploits $\Gamma_{\text{close}}(p)$ to progressively discover the k furthest neighbors from p. The intuition behind these two heuristics (shown in Figure 5.2 in the case of the node *Alice*) is as follows:

- The first heuristic (termed *far-from-close* and labeled **1** in the figure) requests the "far neighborhood" $\Gamma_{\text{far}}(B)$ of a node *Bob* found in *Alice*'s "close neighborhood"

$\Gamma_{\text{close}}(A)$. The idea is that if *Bob* is close to *Alice*, then nodes that are far from *Bob* (such as *Carl* in Figure 5.2) will also be far from *Alice*.

- The second heuristic (termed *close-to-far* and labeled **2** in the figure) requests the "close neighborhood" $\Gamma_{\text{close}}(D)$ of a node *Dave* found in *Alice*'s "far neighborhood" $\Gamma_{\text{far}}(A)$. The idea is that if *Dave* is far from *Alice*, then nodes that are close to *Dave* (such as *Ellie* in Figure 5.2) will also be far from *Alice*.

In the following we present HyFN, a general algorithm that combines the two heuristics described above in various measures.

5.3 Algorithms

5.3.1 General Framework

Algorithm 10 provides an overview of the approach we propose, termed HyFN, as executed by Node p. For a fair comparison with a traditional greedy approach, we limit ourselves to *one* push-pull exchange per round and per node (as in Algorithm 9). This limitation is key to properly assess the interest of our approach: an algorithm that exchanges more information is naturally advantaged against its more frugal competitors. It would for instance be unfair to compare an algorithm using multiple push-pull exchanges to maintain multiple views against Algorithm 9, as such an algorithm would be more costly in terms of network traffic.

To ensure only one push-pull exchange is performed per round we use the construct **with probability** α **do** .. **otherwise** at line 3. This construct executes with a given probability (here α) the first statement, and with a probability $(1 - \alpha)$ the second. In this particular case, Algorithm 10 randomly alternates between invoking updateCloseView() at line 4, and invoking updateFarView() at line 6. Both procedures (discussed below), only generate one network exchange per node and per round, thus enforcing our communication limit. updateCloseView() maintains $\Gamma_{\text{close}}(p)$, p's close neighborhood, while updateFarView() uses $\Gamma_{\text{close}}(p)$ to construct $\Gamma_{\text{far}}(p)$. The parameter α (contained in $[0, 1]$) measures out how much effort each node will spend on $\Gamma_{\text{close}}(p)$ rather than $\Gamma_{\text{far}}(p)$.

updateCloseView(), shown at lines 7-11, uses Algorithm 9 (discussed in Section 5.2.1) to construct $\Gamma_{\text{close}}(p)$. updateFarView() depends on a pluggable procedure farCandidatesXX(p), which exchanges potential new candidate nodes using a push-pull approach to update p's far neighborhood, $\Gamma_{\text{far}}(p)$ at line 16. The current far neighborhood of p, the nodes received by farCandidatesXX(p), and r random nodes are stored in the intermediate $cand_{\text{far}}$ variable (line 16). The k furthest nodes from $cand_{\text{far}}$ then become p's new far neighborhood (line 17; note the minus sign before $\text{sim}(p, g)$, in contrast to line 11). (We discuss the push part of the exchange just below.)

Algorithm 10: HyFN: A generic algorithm to implement a KFN computation, executing at node p

```
1  Init: For each p, Γ_close(p) and Γ_far(p) are heaps of size k, initialized as empty.
2  each round do
3      with probability α do
4          updateCloseView()
5      otherwise
6          updateFarView()
7  procedure updateCloseView() is
8      q ← one random neighbor from Γ_close(p)
9      send ⟨close, Γ_close(p) ∪ {p}⟩ to q ; request Γ_close(q) from q   ▷ push-pull
10     cand_close ← Γ_close(p) ∪ Γ_close(q) ∪ {r random nodes} \ {p}
11     Γ_close(p) ← argtop^k_{g∈cand_close} (sim(p, g))
12 on receiving ⟨close, Γ'_close⟩ do
13     cand_close ← Γ_close(p) ∪ Γ'_close \ {p}
14     Γ_close(p) ← argtop^k_{g∈cand_close} (sim(p, g))
15 procedure updateFarView() is
16     cand_far ← Γ_far(p) ∪ farCandidatesXX(p) ∪ {r random nodes}
17     Γ_far(p) ← argtop^k_{g∈cand_far} ( − sim(p, g))
```

5.3.2 Instantiating the selection of far candidates

The pluggable method farCandidatesXX(p) can be instantiated in three different manners, with the procedures farCandidatesFarFromClose(p), farCandidatesCloseToFar(p) and farCandidatesMixed(p), shown in Algorithms 11, 12, and 14.

- farCandidatesFarFromClose(p) (Algorithm 11) implements the *far-from-close* strategy discussed in Section 5.2.2: the local node p first selects one of its close neighbors q_{close} (line 2), and returns the far neighbors of q_{close}, $\Gamma_{\text{far}}(q_{\text{close}})$,

Algorithm 11: A far-from-close strategy to select far candidates (at p)

```
1  procedure farCandidatesFarFromClose(node p) is
2      q_close ← one random neighbor from Γ_close(p)
3      send ⟨far, Γ_far(p)⟩ to q_close ; request Γ_far(q_close) from q_close   ▷ pull
4      return Γ_far(q_close)
```

Algorithm 12: A close-to-far strategy to select far candidates (at p)

```
1 procedure FARCANDIDATESCLOSETOFAR(node p) is
2     q_far ← one random neighbor from Γ_far(p)
3     send ⟨FAR, Γ_close(p) ∪ {p}⟩ to q_far ; request Γ_close(q_far) from q_far   ▷ pull
4     return Γ_close(q_far)
```

Algorithm 13: Reception of a FAR push message (at p)

```
1 on receiving ⟨FAR, Γ'_far⟩ do
2     cand_far ← Γ_far(p) ∪ Γ'_far
3     Γ_far(p) ← argtop^k_{g∈cand_far} ( − sim(p, g))
```

as new candidates to update $\Gamma_{\text{far}}(p)$. In addition, the procedure pushes towards q_{close} the far neighbors of p, as nodes far from p are likely to lay far from q_{close} as well. The receipt of the corresponding FAR message is handled by the code shown in Algorithm 13.

- FARCANDIDATESCLOSETOFAR(p) (Algorithm 12) implements the *close-to-far* strategy presented above: this time, p picks one of its current far neighbors q_{far}, and returns the close neighbors of q_{far}, $\Gamma_{\text{close}}(q_{\text{far}})$ in order to improve $\Gamma_{\text{far}}(p)$. The procedure also pushes towards q_{far} the close neighborhood of node p, $\Gamma_{\text{close}}(p)$, as those are likely to lay far from q_{far}. The push message, of type FAR, is handled as above.
- FARCANDIDATESMIXED(p) (Algorithm 14) combines the two above strategies in one single heuristics. As in Algorithm 10, we use the **with probability** construct to switch between the *far-from-close* and *close-to-far* strategies with probability β, thus insuring that only one push-pull exchange occurs every time FARCANDIDATESMIXED(p) is invoked. The parameter β further controls how much each strategy is used, and allows FARCANDIDATESMIXED(p) to generalize the previous two procedures: the extreme case $\beta = 0$ corresponds to the *far-from-close* strategy, while $\beta = 1$ implements a *close-to-far* approach.

Considered all-together, Algorithms 10 to 14 capture a family of decentralized k-

Algorithm 14: A mixed strategy to select far candidates (at node p)

```
1 procedure FARCANDIDATESMIXED(node p) is
2     with probability β do
3         return FARCANDIDATESCLOSETOFAR(p)
4     otherwise
5         return FARCANDIDATESFARFROMCLOSE(p)
```

furthest-neighbor (KFN) graph construction protocols, controlled by two stochastic parameters, α and β. Parameter α controls the distribution of efforts between the intermediate KNN view and the final KFN view, while β arbitrates between the *far-from-close* and *close-to-far* strategies.

Note that some gossip protocols, such as the original T-Man, tailor the candidates they send to the specific node that requested them, while we do not. For instance, in FARCANDIDATESFARFROMCLOSE, q sends back the same set $\Gamma_{\text{far}}(q)$ as potential new neighbors for p, whatever node p sent the request. This set is not tailored to a specific node p. This is because those other protocols work with an unbounded view that keeps all data received but fixed-size messages, and so they want to send back the best information they have available. As our approach works with fixed-size view, we simply send the full set of node.

5.4 Evaluation

We evaluate our framework using the simulator PeerSim [61], and compare its behavior against a basic greedy epidemic protocol (Algorithm 9) that uses a negative similarity metric (Equation 5.1). We term this baseline solution *Far From Far* and we note that this is strictly better than taking purely random nodes: it selects the best neighbors from candidates specifically including random nodes from the peer-sampling service, but also some additional nodes known from one-hop neighbors.

We are essentially interested in two aspects of our solution: (i) *its convergence*, i.e. how fast our framework is able to converge to a good KFN graph, and (ii) *its scalability*, i.e. how does this convergence speed evolve with growing network sizes. The code used for our experiments can be found on-line at `https://gitlab.inria.fr/ASAP/HyFN`.

5.4.1 Experimental set-up and metrics

Unless stated otherwise our default set-up involves 3200 nodes regularly positioned on a $[0, 1)$ ring. By default, we use views of $k = 14$ nodes, and fetch $r = 3$ random nodes in each round; these values were determined empirically and are the smallest possible that still give satisfying performances. We set the parameters of HyFN to $\alpha = \beta = 0.5$. These values mean that on average nodes spend the same number of rounds constructing their KNN and KFN views (α at line 3 of Algorithm 10), and that the construction of the KFN view uses the heuristics *far-from-close* and *close-from-far* in equal measure (β at line 2 of Algorithm 14). We assume a random peer sampling service (RPS) [43] is available, which we use to initialize all views with random nodes before the protocol starts, and to provide r random nodes in each round.

To measure the convergence of the approximate KFN graph constructed by HyFN we use the following four metrics:

- **Number of missing links**: We count for each node how many of its k furthest neighbors are missing from its KFN view. The count of all these *missing links* over the network yields our first metric.
- **Number of converged nodes**: As a second measure of convergence, we consider that a node is converged when at least 80% of its k furthest neighbors (taking into account ties) are contained in its KFN view. As a measure of the network's convergence, we count in each round how many nodes are converged.
- **Average KFN distance**: For each node, we compute the average distance between this node and the nodes in its KFN view. This metric should tend toward 0.5 in a ring of perimeter 1 (our default topology), so we re-normalize our results to be between 0 and 1 and call this the **average similarity** in the various graphs below. Note that even a perfectly converged network won't actually reach 1 though, with the exact value depending on the density of the network; with 3200 nodes, the difference is not visible on graphs.
- **Convergence time** Finally, we consider that the whole network is converged when at least 80% of all nodes are converged, according to the above criterion. We count the number of rounds until this convergence condition is fulfilled.

We do not report the communication overhead of either HyFN or our baseline: the protocols are all designed to initiate one single push-pull exchange in each round, and therefore present exactly the same communication costs, no meaningful comparison is possible.

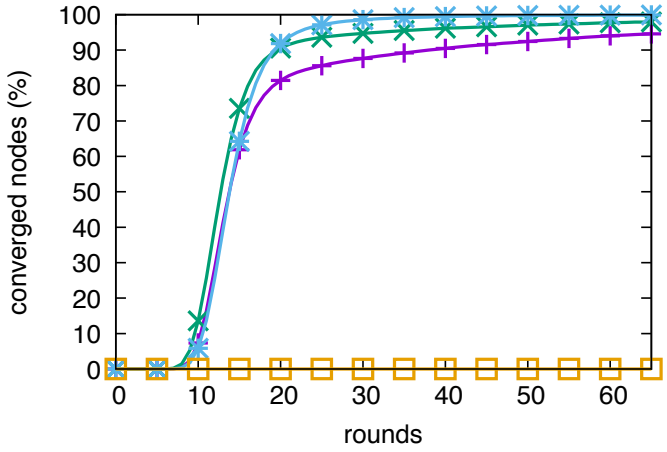
In the following we first evaluate HyFN on our default scenario (3200 nodes on a regular ring, $k = 14$, $r = 3$, $\alpha = \beta = 0.5$, the values for k and r being the smallest values still providing functional results) and compare it against our baseline. We then analyze the impact of the mixing parameters α and β. Finally, we study the scalability of HyFN up to networks of 12800 nodes, both on a ring and grid topology. All reported values are averages computed over 25 experimental runs.

5.4.2 Results

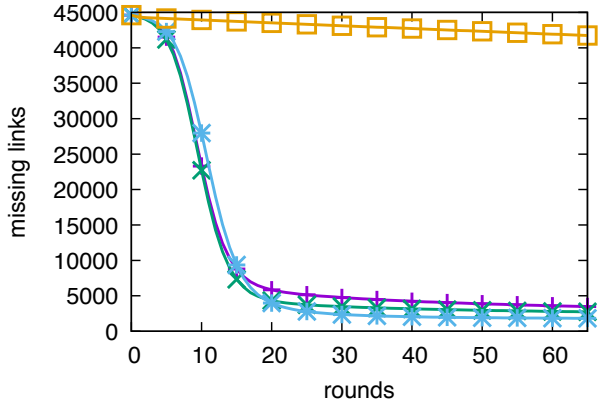
Figure 5.3 shows the convergence of HyFN in our default scenario (3200 nodes on a regular ring), according to three convergence metrics: the percentage of converged nodes (Figure 5.3a), the number of missing links (Figure 5.3b), and the average KFN similarity (normalized to 1, Figure 5.3d). The behavior of three variants of HyFN are shown, which correspond to the three heuristics presented in Algorithms 11 (*Far-from-Close*), 12 (*Close-to-Far*), and 14 (*Hybrid*), discussed in Section 5.3.2.

Comparison to the Far-from-Far baseline.

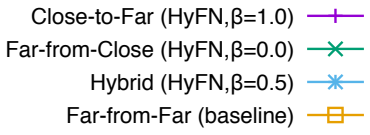
From the three convergence metrics, it appears that the three versions of HyFN clearly outperform the baseline. More precisely, all HyFN variants have reached 80% of



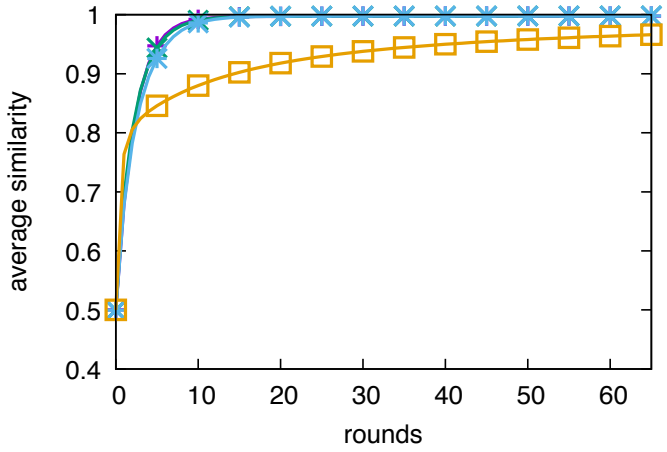
(a) Converged nodes



(b) Missing links



(c) Legend



(d) Average similarity

Figure 5.3: Converged nodes, missing links, and average similarity for the baseline (*Far-from-Far*) and for three versions of HyFN (corresponding to $\beta = 1$ for *Close-to-Far*, $\beta = 0$ for *Far-from-Close* and $\beta = 0.5$ for *Hybrid*) on a 3200-node regular ring. In all cases, all three versions of HyFN significantly outperform the baseline, with the hybrid approach ($\beta = 0.5$) being the optimal trade-off.

converged nodes after at most 20 rounds whereas the baseline is unable to converge even after 65 rounds (Figure 5.3a). Interestingly, the hybrid variant has the best performances in terms of overall convergence. From the average similarity metric (Figure 5.3d), the baseline has the worst performances, even if it gets decent results in a reasonable time. In fact, it doesn't get the farthest neighbors, but still it gets far neighbors. Moreover, the metric of missing links (Figure 5.3b) shows clearly that the baseline does not work: it just converges linearly only due to the couple of random neighbors that are fetched at each turn. Finally, among all HyFN variants, the *Hybrid* approach seems to converge most closely to the theoretically ideal network at the price of being a slightly slower than *Close-to-Far*.

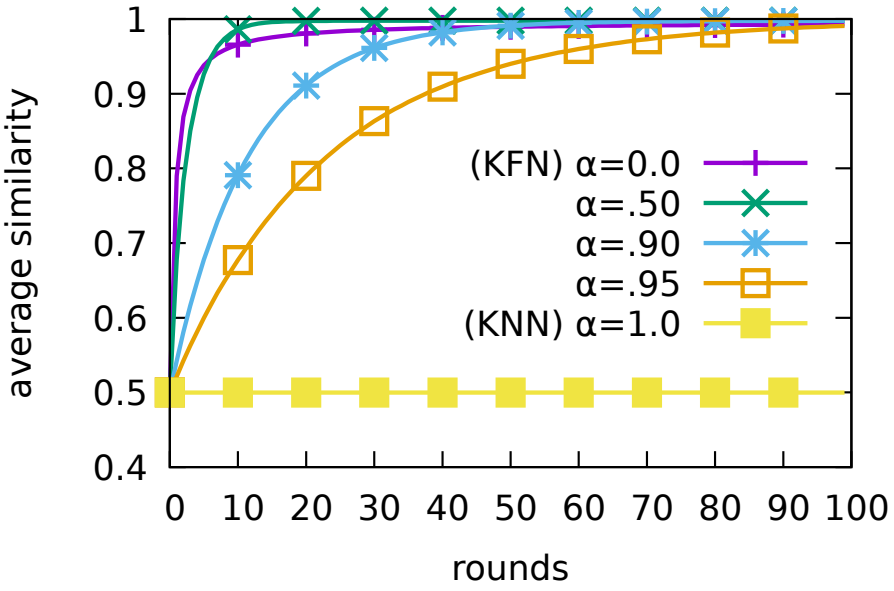
Influence of the parameters α and β.

Our key aim is to evaluate the effective impact of the stochastic parameters α and β on the KFN graph and to set them accordingly. Figure 5.4 outlines the impact of the α parameter, and shows that $\alpha = 0.5$ is close to the optimal. This value provides: (i) the best convergence time (Figure 5.4b), and (ii) the best tradeoff between the convergence speed and the quality of the neighborhood (Figure 5.4a). Concerning the impact of fine tuning β (Figure 5.5), having β close to 0.2 gives the best network convergence, and convergence speed. Note that we are not able to reach 100% of converged nodes when we choose a β value of either 0 or 1. As a result having a non hybrid heuristic is not the most suitable choice, although the results of these kind of heuristics is still better than the baseline. Furthermore, as soon as we use the hybrid strategy, the value of $0 < \beta < 1$ has a little impact on the convergence time.

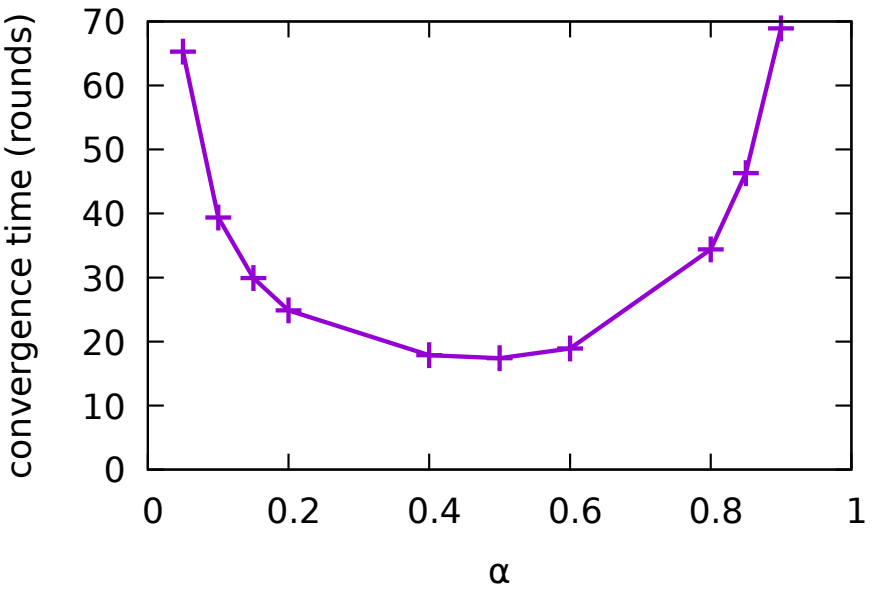
Consequently, it appears that fine tuning α is predominant compared to β. In other terms, once we have set α to its best value (i.e 0.5), the value of β has a little impact as long as $0 < \beta < 1$, so as long as we are actually using an hybrid approach.

Scalability.

We have investigated the applicability of the hybrid heuristic on both a ring and grid logical networks of varying sizes from 100 to 12800 nodes (Figure 5.6). The values for k and r in the default 3200-node configuration where the smallest possible while still providing good performances, and it is a known property that these parameters evolve logarithmically with respect to the size of the network s. So for every configuration, we set up $k = 1.2 * \log_2(s)$ and $r = 0.3 * \log_2(s)$, both rounded to the closest integer — in order to get back $k = 14$ and $r = 3$ for $s = 3200$. As a result, it appears that HyFN converges as expected in logarithmic time relative to the network total size, demonstrating thus that our approach scales well.

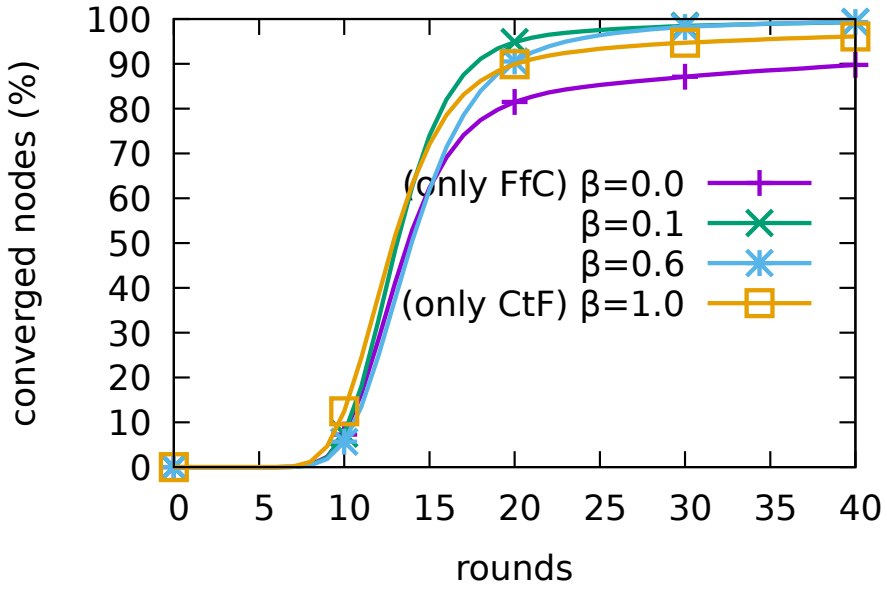


(a) Convergence pace for various values of α. For $\alpha = 1$, we are only updating the KNN view, so the average similarity stagnates at 0.5, due to the uniform random initialization which is never updated. For $\alpha = 0$, we only update the KFN view, so the similarity improves very quickly at the beginning but gets stuck in local maxima and takes a very long time reaching a fully converged state. Other values for $0 < \alpha < 1$ offer various trade-offs between initial speed and full convergence time, with the optimal value at $\alpha = 0.5$.

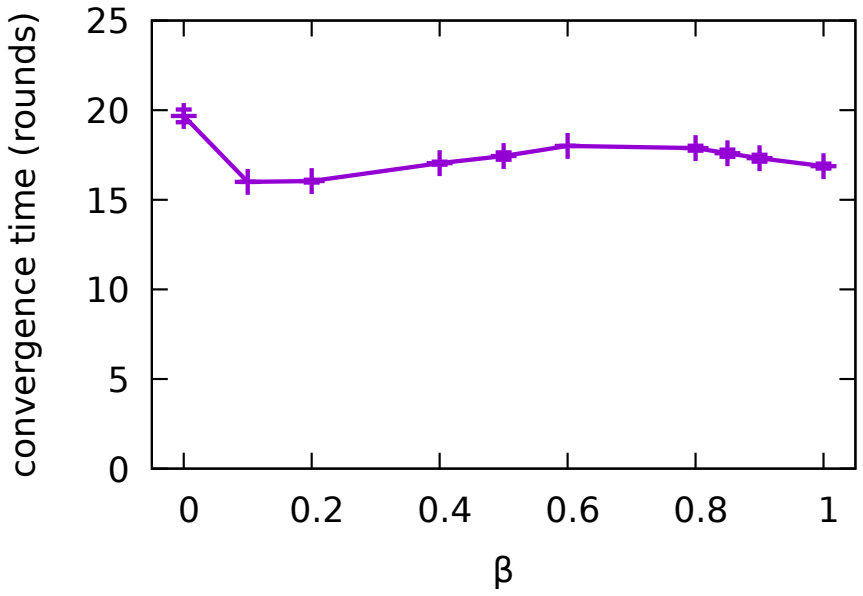


(b) Convergence time for various value of α. The optimal value is clearly at $\alpha = 0.5$, with performances regularly degrading as we go closer to 0 or 1.

Figure 5.4: Impact of the α stochastic parameter on a 3200-node regular ring.



(a) Convergence pace for various values of β. For $\beta = 0$ and $\beta = 1$, we never reach a 100% converged state. Other values are very close to each other.



(b) Convergence time for various values of β. Optimal value is approximately $\beta = 0.2$, but the overall impact is small.

Figure 5.5: Impact of the β stochastic parameter on a 3200-node regular ring.

5.5 Conclusion

In this chapter, we have proposed HyFN, a novel and generic decentralized protocol to compute k-furthest-neighbor (KFN) graphs. HyFN exploits an intermediate k-nearest-neighbor (KNN) graph, which is constructed in parallel, to progressively con verge towards an optimal solution. We have in particular proposed three heuristics to

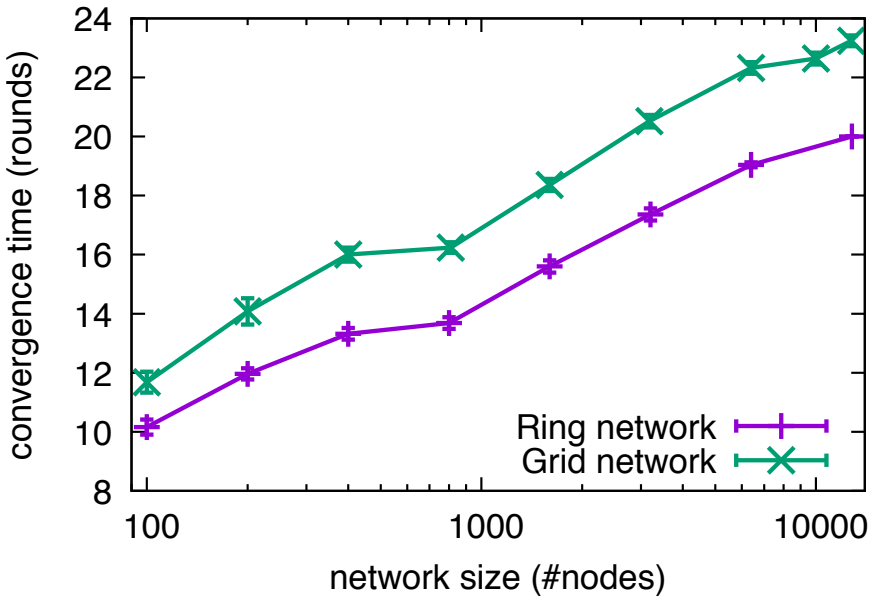


Figure 5.6: Behavior of HyFN with the hybrid heuristic for networks from s = 100 nodes to s = 12800 nodes, for a variety of network topologies (Ring and Grid in the above figure).

exploit this KNN graph. Our evaluation shows that our proposal clearly outperforms a naive greedy implementation based on existing KNN epidemic protocols.

Beyond its application to decentralized and pair-to-pair systems, we believe our KFN construction framework holds a strong potential for the computation of KFN graphs on highly parallel machines. Its inherent properties of locality and high concurrency are likely to make it a worthwhile approach in cases in which a KFN graph is required, including resource allocation problems such as those encountered in VM allocation services.

More generally, HyFN is evidence that epidemic protocols are easy to combine and extend to new domains, even where some of the basic hypothesis do not hold anymore, and can cover a very wide range of applications. As such, they are indeed good candidates to implement smarter basic blocks with self-adaptive mechanisms for opportunistic systems.

In the next chapter, we will conclude this thesis with a summary of our various contributions, a discussion of future works opened by our projects and our general perspective on holistic approaches for self-adaptive opportunistic systems.

Chapter 6

Conclusion

Nowadays, with the advent of the Internet of Things and other large scale distributed systems such as Smart Cities, or Self-Driving Fleets, distributed systems are becoming increasingly pervasive and complex, with systems involving very heterogeneous components, dozens to hundreds of separate components composing a single system.

Consequently, modern distributed systems are intractable, and we posit the need for a more holistic approach, with higher-level abstractions, to consider a system's function as a whole, away from the behavior of individual nodes and parts.

Moreover, to go with those new abstractions and hide the growing complexity of distributed ecosystems, basic blocks need to be smart enough to react to an evolving environment and changing circumstances, including failures, in order to compose **self-adapting and opportunistic systems**, that are easier to handle at all stages of a system's life: design, implementation, deployment, and maintenance.

6.1 Summary of contributions

In this thesis, we have proposed three contributions that seek to progress towards this high-level ideal vision.

Pleiades In Chapter 3, we presented Pleiades, a holistic approach to the construction and maintenance of complex distributed structures. Pleiades combines two long-running concepts of distributed computing and software engineering: *self-organizing overlays* and *programming by assembly*. The resulting approach allows developers to describe programmatically complex topologies as an assemblage of simpler shapes. This description is then instantiated on a population of available nodes by a gossip-based run-time engine that handles node-to-node communication and other low-level details, thanks to a collection of concurrent and collaborating self-stabilizing decentralized protocols. The resulting topologies are scalable, highly resilient in the face of failures, and lightweight, thus offering a number of attractive properties to today large-scale distributed systems. Pleiades demonstrates that combining higher-level

abstractions and smarter basic blocks is indeed a viable approach to make distributed systems tractable again while hiding the growing complexity.

Mind-the-Gap In Chapter 4, we proposed Mind-the-Gap, a gossip-based protocol able to detect partitions (and other large connectivity changes) in MANETs, a central issue users of these environments must address. Our approach relies on an opportunistic aggregation strategy that constructs a stochastic representation of the network's current composition, and uses this approximate knowledge to detect large membership changes. It shows how randomized structures can provide lightweight yet robust services that enable highly dynamic decentralized systems to be aware of their environment and detect changes in their circumstances, even using only local information, a first step toward opportunistic self-adapting systems. We think it opens interesting research paths for highly distributed application domains such as the Internet of Things, Smart Cities, and Self driving fleets.

HyFN In Chapter 5, we have explored how gossip protocols could be extended to work on a more diverse set of problems, notably in cases where nodes should be grouped according to dissimilarity rather than sameness. This k-Furthest-Neighbors problem is typically hard for traditional approaches which relies on an implied transitivity in the neighborhood relationship, but our proposal, HyFN a two-layer hybrid approach, demonstrates how the parallel composition of decentralized self-organizing protocols can be exploited to deliver a richer functionality and better convergence properties at no additional cost. It highlights that gossip protocols are robust and easy to combine and extend to new applications, thus making them promising candidates to realize the low-level maintenance needed for modern distributed systems through self-organizing overlays.

Overall, our work has shown that the vision we proposed in Chapter 1 is a realistic and promising direction for future research.

6.2 Future research directions

The work realized in this thesis has opened new research directions. Indeed, the vision described in this dissertation introduction is obviously far from completed yet, but we believe our results have confirmed its potential. The next step would be to extend and generalize our work to less specific contexts.

Holistic approaches Starting from Pleiades, there are two distinct possibilities to generalize our approach. The first one is to work on the interface between the topology and the applications running on top of it. Indeed, Pleiades showed it is

possible to build complex topologies in a simple and efficient way thanks to proper abstractions, but the real value of a topology is to support and enable an application with certain properties. For instance, a ring with fingers has no immediate value, but in Chord it allows distributed storage with efficient retrieval time. PLEIADES still lacks the necessary support to bridge that gap between topology and function, and designing and implementing a proper API to connect the two, that is easy to use and yet generic enough to work in a large number of contexts and support our vision would seem particularly interesting.

The second possibility would be to go from combining shapes in order to create topology, to combining functions in order to create more complex features. Focusing specifically on topology allowed us to model the interface between our basic blocks quite simply with the generic "ports" we described in Chapter 3. Extending this process to functions would provide a bridge to recent efforts around serverless infrastructures and Function-as-a-Service platforms such as AWS Lambda. We conjecture this would require extensive work to precisely define a taxonomy with the basic functions of a distributed system, along with their inputs and outputs. We would also need to add a notion of dependency, possibly with multiple levels of importance, from critical requirements necessary to operate the system, to optional components which only ensure a better Quality of Service.

Another independent aspect that could be improved is the usability and programmability of our approach. Our prototype and simulations demonstrated that a wide range of complex topologies could be described with a relatively small number of primitives, but a designer still needs to tinker with various configuration files and fiddle manually with parameters. Creating a proper configuration Domain Specific Language (DSL) and the tool-chain to go with it not only would let designers tackle more complex tasks more easily, but would also force us to refine and extend our primitives.

Opportunistic systems Following MIND-THE-GAP, we raised two independent questions:

(i) How to detect a larger class of events, and generally increase a system awareness of its own environment?
(ii) How to move forward from autonomous detection to self-adaptation, without manual intervention from an operator?

For the first part, there is probably no way to work around the fact we will need a different detector for each class of event, but the important part of the work would be to study what a large-scale decentralized system needs to know about its environment, what kind of events it needs to detect, and which ones are important enough to justify the cost of monitoring, both in terms of added complexity to the system, and of resource usage. Ultimately, the monitoring sub-system itself likely needs to be self-

adapting, regulating its own resources and watching for more events when the risk is higher.

For the second part, the solution almost certainly involves machine-learning in some way, but adapting machine-learning methods to distributed systems is a tricky proposition. Indeed, training is a resource intensive process that requires a lot of computational power and large amount of data, two ingredients that are difficult to obtain in systems where individual nodes generally have only limited capabilities and local knowledge. Resorting to training in centralized traditional infrastructure poses different challenges, such as gathering the data from the deployed systems and disseminating the results back. There are also ethical issues, with Smart Home or Self-Driving Cars handling private and sometimes very sensitive information, which may be inappropriate to disclose for training.

6.3 Perspective

Beyond our technical contributions, we want to offer a few more general insights.

Theoretical work, practical consequences: Our first insight is that abstractions and models are important in practice, not just idle theorizing from scientists in their ivory tower. Programmers may not care about the theoretical aspect, but providing them with a mental framework to manipulate complex notions do get things done, in a more efficient and less error-prone manner.

Various related domains such as Programming Languages and Software Engineering have already gone through a similar process, and can in this respect serve as a continuous source of inspiration. This is work that can be slow to trickle down, but it is important nonetheless to engage in it, similarly to how communities develop best-practices and common habits to facilitate communication and cooperation.

Furthermore, because of their rapidly evolving nature, distributed systems remain a developing field and require the repeated application of this process. We believe that most notably with the advent of the Internet of Things and Smart Cities, distributed systems are entering a new development phase and have to get that work done for those new contexts, with an holistic approach to these issues.

Basic yet Smart: The second point we want to stress is that the scale and complexity of distributed systems is exploding. Even with better and higher-level abstractions, this won't be enough to compensate, and managing everything manually will become strictly intractable in the short to medium term. Consequently, it is crucial when designing modern distributed systems to make sure the basic blocks are somewhat autonomous. They must be able to detect changes or errors, react to and correct them, adapt and evolve over time. In other words, the basic building blocks must be smart enough to alleviate developers and automate most of the low level work, with

self-configuration and robust interfaces at the boundaries between systems, enabling opportunistic collaborations.

Gossip is sick, mate: Third, we want to promote gossip and epidemic protocols. In their basic form, they are already teeming with many desirable properties: rapid dissemination/gathering of information, quick convergence, natural resilience to nodes crashing, fully decentralized and able to work with only local knowledge, no single point of failure, and so on. But on top of that, they are also easy to extend and combine and even simple greedy iterative optimizations can be creatively stacked to realize complex higher-level functions, and their field of application is extremely wide. Simultaneously, the rise of new fields requiring a large number of loosely coupled entities such as the IoT provides, we believe, an excellent opportunity to further develop and apply this family of protocols.

Everything is political: Finally, we want to conclude this scientific dissertation with an entirely non-scientific yet very important message: technique is not everything. The best algorithm ever, even with perfect performances, still need to be recognized as such, and this is a social and political issue. Standardization, backward compatibility with legacy software, and maturity are just some of many crucial issues impacting the transfer of new ideas to practitioners and industry. In particular, if holistic approaches are to be widely adopted, they will probably need a common "platform" to help capitalize on past results and pool development effort. Free software (or at least, open source code) and open standards will have a critical role in providing this kind of positive loop and in the creation of a healthy distributed ecosystem, to enable cooperation on a very large scale.

Also, opening further towards wider social concerns, remember that "code is law"[1], and as distributed systems enter more deeply into our lives, we expect their ethical aspects to become increasingly important. We would argue that Privacy and Security concerns should be taken into account as first class issues from the design stage up, especially when pervasive distributed systems will have access to every detail of their users' everyday life, or will control critical infrastructures such as roads, water distribution or electrical power grid. This will be a fascinating challenge, but not one that computer scientists should tackle alone, and we need inter-disciplinary work and collaborations with law experts, regulators, and industrials, but also scientists from other fields such as Sociology or Economy.

[1]Lawrence Lessig, *Code is Law – On Liberty in Cyberspace*, https://harvardmagazine.com/2000/01/code-is-law-html

Acknowledgment

This thesis wasn't a solitary work, and over all those years I had the privilege to receive a lot of help from very many great people.

First of all, I wish to express my deep gratitude to my thesis directors Prof. François Taïani and Prof. David Bromberg for their patience and their thoughtful guidance. This would have never been possible without them.

More generally, I would like to thank all the researchers that mentored me, during and before my thesis, and who helped me grow, both as a scientist and as a person. There are too many of them to cite them all, but they are not forgotten. A special mention to Prof. Krishna Gummadi who deepened and reinforced my conviction that good science is socially conscious and engaged in the real world.

Of course, many thanks to all the jury members who accepted to be a part of my work's evaluation. Academia involves a lot of volunteer service for the good of the community, and I am very grateful for the time they gave me. I'll do my best to repay it forward. I am especially grateful to professors Paulo Ferreira and Philippe Lamarre for reporting on this work.

A very special thank to Virginie Desroches and Cécile Bouton, our team assistants, who are too often forgotten and whose tireless work make ours run smoothly. It's too easy to notice only when things go bad, but you have gone beyond the call of duty so please know that your support was invaluable over those years.

To all my friends and colleagues from the ASAP/WIDE team, a big thank you for your company, your advices, and all those lunch-and-coffee-break heated but friendly debates. You made work a bit more fun. Special mention to my office-mates who tolerated my prattling even outside of breaks.

But life isn't all work, so finally, my heartfelt thoughts go to my various flat-mates over the year. Thanks for making sure I still had a social life and taking care of me when I didn't do it myself.

Bibliography

[1] K. Akkaya and M. Younis. A survey on routing protocols for wireless sensor networks. *Ad Hoc Networks*, 3(3):325 – 349, 2005.

[2] L. Arantes, P. Sens, G. Thomas, D. Conan, and L. Lim. Partition participant detector with dynamic paths in mobile networks. In *9th IEEE Intl. Symp. on Network Computing and App.*, NCA, 2010.

[3] X. Bai, M. Bertier, R. Guerraoui, A.-M. Kermarrec, and V. Leroy. Gossiping personalized queries. In *EDBT'2010*.

[4] J.-P. Banâtre, P. Fradet, and Y. Radenac. Principles of chemical programming. In *5th Int. Workshop on Rule-Based Programming*, volume 124(1) of *ENTCS*, pages 133–147. Elsevier, June 2005.

[5] 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.

[6] R. Barazzutti, P. Felber, H. Mercier, E. Onica, and E. Rivière. Efficient and confidentiality-preserving content-based publish/subscribe with prefiltering. *IEEE Trans. Dependable Sec. Comput.*, 14(3):308–325, 2017.

[7] P. Barooah, H. Chenji, R. Stoleru, and T. Kalmár-Nagy. Cut detection in wireless sensor networks. *IEEE Transactions on Parallel and Distributed Systems*, 23(3):483–490, 2012.

[8] I. Bekmezci, O. K. Sahingoz, and S. Temel. Flying ad-hoc networks (FANETs): A survey. *Ad Hoc Networks*, 11(3), 2013.

[9] A. Beloglazov, J. Abawajy, and R. Buyya. Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing. *FGCS*, pages 755–768, 2012.

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

[11] 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.

[12] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communication of the ACM*, 13(7):422–426, 1970.

[13] 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.

[14] S. Bouget, Y.-D. Bromberg, A. Luxey, and F. Taïani. Pleiades: Distributed structural invariants at scale. In *DSN 2018 – IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 542–553, 2018.

[15] S. Bouget, Y.-D. Bromberg, H. Mercier, É. Rivière, and F. Taïani. Mind the Gap: Autonomous detection of partitioned MANET systems using opportunistic aggregation. In *SRDS 2018 – 37th IEEE International Symposium on Reliable Distributed Systems*, pages 143–152, 2018.

[16] S. Bouget, Y.-D. Bromberg, F. Taïani, and A. Ventresque. Scalable Anti-KNN: Decentralized computation of k-Furthest-Neighbor graphs with HyFN. In *DAIS 2017 – IFIP International Conference on Distributed Applications and Interoperable Systems*, pages 101–114, 2017.

[17] 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.

[18] A. Boutet, D. Frey, R. Guerraoui, A. Jégou, and A.-M. Kermarrec. WhatsUp Decentralized Instant News Recommender. In *IPDPS*, 2013.

[19] A. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. *Internet Mathematics*, 1(4):485–509, 2002.

[20] 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.

[21] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes. Borg, omega, and kubernetes. *Communications of the ACM*, 59(5):50–57, 2016.

[22] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, Mar. 1996.

[23] G. Chockler, R. Melamed, Y. Tock, and R. Vitenberg. Spidercast: a scalable interest-aware overlay for topic-based pub/sub communication. In *DEBS*, pages 14–25, 2007.

[24] 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.

[25] D. Conan, P. Sens, L. Arantes, and M. Bouillaguet. Failure, disconnection and partition detection in mobile environment. In *7th IEEE International Symposium on Networking Computing and App.*, NCA, 2008.

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

[27] 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).

[28] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic Algorithms for Replicated Database Maintenance. In *PODC'87*.

[29] 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.

[30] K. Fall. A delay-tolerant network architecture for challenged internets. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM, 2003.

[31] D. Frey, A. Jégou, and A.-M. Kermarrec. Social Market: Combining Explicit and Implicit Social Networks. In *SSS'11*, Grenoble, France, Oct. 2011. LNCS.

[32] D. Frey, A.-M. Kermarrec, C. Maddock, A. Mauthe, P.-L. Roman, and F. Taïani. Similitude: Decentralised adaptation in large-scale P2P recommenders. In *IFIP DAIS'15*, pages 51–65, Grenoble, France, 2-4 June 2015.

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

[34] 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.

[35] P. Grace, D. Hughes, B. Porter, G. S. Blair, G. Coulson, and F. Taiani. Experiences with open overlays: A middleware approach to network heterogeneity. In *European Conf. on Comp. Sys.*, EuroSys, 2008.

[36] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics*. Addison-Wesley, 1994.

[37] 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.

[38] Z. Han, A. L. Swindlehurst, and K. R. Liu. Optimization of MANET connectivity via smart deployment/movement of unmanned air vehicles. *IEEE Trans. on Vehicular Tech.*, 58(7):3533–3546, 2009.

[39] M. Hauspie, J. Carle, and D. Simplot. Partition detection in mobile ad-hoc networks using multiple disjoint paths set. In *International Workshop on Objects models and Multimedia technologies*, 2003.

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

[41] M. Jelasity, A. Montresor, and Ö. Babaoglu. Gossip-based aggregation in large dynamic networks. *ACM TOCS*, 23(3), 2005.

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

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

[44] 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.

[45] D. Kempe, A. Dobra, and J. Gehrke. Gossip-based computation of aggregate information. In *44th Annual IEEE Symposium on Foundations of Computer Science*, FOCS, 2003.

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

[47] A.-M. Kermarrec and F. Taïani. Diverging towards the common good: heterogeneous self-organisation in decentralised recommenders. In *SNS'2012*.

[48] A. Khelil, P. J. Marrón, C. Becker, and K. Rothermel. Hypergossiping: A generalized broadcast strategy for mobile ad hoc networks. *Ad Hoc Networks*, 5(5):531–546, 2007.

[49] A. Khelil, P. J. Marrón, R. Dietrich, and K. Rothermel. Evaluation of partition-aware manet protocols and applications with ns-2. In *Intl. Symp. on Performance Evaluation of Computer and Telecommunication Systems*, SPECTS, 2005.

[50] J. Koomey and J. Taylor. New data supports finding that nearly a third of capital in enterprise data centers is wasted, 2015.

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

[52] X. Li, A. Ventresque, J. O. Iglesias, and J. Murphy. Scalable correlation-aware virtual machine consolidation using two-phase clustering. In *HPCS*, pages 237–245, 2015.

[53] 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.

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

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

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

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

[58] B. Milic, N. Milanovic, and M. Malek. Prediction of partitioning in location-aware mobile ad hoc networks. In *38th Annual Hawaii International Conference on System Sciences*, HICSS, 2005.

[59] 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.

[60] 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/.

[61] A. Montresor and M. Jelasity. PeerSim: A scalable P2P simulator. In *P2P'09*, 2009.

[62] 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.

[63] 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.

[64] R. Motwani and P. Raghavan. *Randomized algorithms*. Chapman & Hall/CRC, 2010.

[65] 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.

[66] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *SIGCOMM '01*, pages 161–172, New York, NY, USA, 2001. ACM.

[67] H. Ritter, R. Winter, and J. Schiller. A partition detection system for mobile ad-hoc networks. In *1st IEEE ComSoc Conference on Sensor and Ad Hoc Communications and Networks*, SECON, 2004.

[68] A. Rowstron and P. Druschel. *Middleware 2001*, 2001.

[69] P. Ruiz and P. Bouvry. Survey on broadcast algorithms for mobile ad hoc networks. *ACM Computing Surveys*, 48(1), July 2015.

[70] T. Saber, A. Ventresque, I. Brandic, J. Thorburn, and L. Murphy. Towards a multi-objective vm reassignment for large decentralised data centres. In *UCC*, 2015.

[71] 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.

[72] 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.

[73] 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/.

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

[75] 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.

[76] A. Varga and R. Hornig. An overview of the OMNeT++ simulation environment. In *1st Intl. Conf. on Simulation Tools and Techniques for Communications, Networks and Systems*, Simutools, 2008.

[77] 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.

[78] S. Voulgaris and M. v. Steen. Epidemic-style management of semantic overlays for content-based searching. In *Euro-Par 2005 Parallel Processing*, number 3648, pages 1143–1152. Springer Berlin Heidelberg, 2005.

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

[80] 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.

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

[82] M. Won, S. M. George, and R. Stoleru. Towards robustness and energy efficiency of cut detection in wireless sensor networks. *Ad Hoc Networks*, 9(3):249–264, 2011.

[83] F. Xu, F. Liu, H. Jin, and A. V. Vasilakos. Managing performance overhead of virtual machines in cloud computing: A survey, state of the art, and future directions. *Proceedings of the IEEE*, 102(1):11–31, 2014.

[84] C. K. Yeo, B.-S. Lee, and M. H. Er. A framework for multicast video streaming over ip networks. *J. of Network and Comp. App.*, 26(3):273–289, 2003.